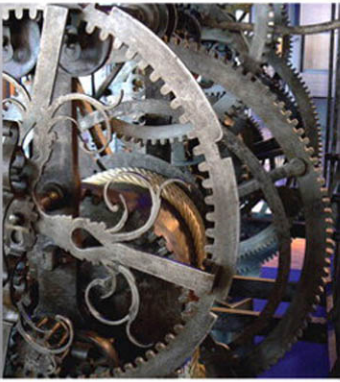
DTrace

Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD



Brendan Gregg • Jim Mauro

Foreword by Bryan Cantrill

DTrace

This page intentionally left blank

DTrace

Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD

Brendan Gregg
Jim Mauro

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Gregg, Brendan.
Dynamic tracing in Oracle Solaris, Mac OS X, and FreeBSD / Brendan Gregg, Jim Mauro.
p. cm.
Includes index.
ISBN-13: 978-0-13-209151-0 (alk. paper)
ISBN-10: 0-13-209151-8 (alk. paper)
1. Debugging in computer science. 2. Solaris (Computer file) 3. Mac OS. 4. FreeBSD. I. Mauro, Jim. II. Title.
QA76.9.D43G74 2011
005.1'4—dc22

2010047609

ISBN-13: 978-0-13-209151-0
ISBN-10: 0-13-209151-8
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, March 2011

Contents

Foreword

In early 2004, DTrace remained nascent; while Mike Shapiro, Adam Leventhal, and I had completed our initial implementation in late 2003, it still had substantial gaps (for example, we had not yet completed user-level instrumentation on x86), many missing providers, and many features yet to be discovered. In part because we were still finishing it, we had only just started to publicly describe what we had done—and DTrace remained almost entirely unknown outside of Sun. Around this time, I stumbled on an obscure little Solaris-based tool called `psio` that used the operating system's awkward pre-DTrace instrumentation facility, TNF, to determine the top I/O-inducing processes. It must be noted that TNF—which arcanely stands for Trace Normal Form—is a baroque, brittle, pedantic framework notable only for painfully yielding a modicum of system observability where there was previously none; writing a tool to interpret TNF in this way is a task of Herculean proportions. Seeing this TNF-based tool, I knew that its author—an Australian named Brendan Gregg—must be a kindred spirit: gritty, persistent, and hell-bent on shining a light into the inky black of the system's depths. Given that his TNF contortionist act would be reduced to nearly a one-liner in DTrace, it was a Promethean pleasure to introduce Brendan to DTrace:

```
From: Bryan Cantrill <bmc@eng.sun.com>
To: Brendan Gregg <brendan.gregg@tpg.com.au>
Subject: psio and DTrace
Date: Fri, 9 Jan 2004 13:35:41 -0800 (PST)
```

```
Brendan,

A colleague brought your "psio" to my attention -- very interesting.
Have you heard about DTrace, a new facility for dynamic instrumentation
in Solaris 10? As you will quickly see, there's a _lot_ you can do with
DTrace -- much more than one could ever do with TNF.
...
```

With Brendan's cordial reply, it was clear that although he was very interested in exploring DTrace, he (naturally) hadn't had much of an opportunity to really use it. And perhaps, dear reader, this describes you, too: someone who has seen DTrace demonstrated or perhaps used it a bit and, while understanding its potential value, has perhaps never actually used it to solve a real problem. It should come as no surprise that one's disposition changes when DTrace is used not to make some academic point about the system but rather to save one's own bacon. After this watershed moment—which we came to (rather inarticulately) call the DTrace-just-saved-my-butt moment—DTrace is viewed not as merely interesting but as essential, and one starts to reach for it ever earlier in the diagnostic process.

Given his aptitude and desire for understanding the system, it should come as no surprise that when I heard back from Brendan again some two months later, he was long past his moment, having already developed a DTrace dependency:

```
From: Brendan Gregg <brendan.gregg@tpg.com.au>
To: Bryan Cantrill <bmc@eng.sun.com>
Subject: Re: psio and DTrace
Date: Mon, 29 Mar 2004 00:43:27 +1000 (EST)

G'Day Bryan,

DTrace is a superb tool. I'm already somewhat dependent on using it.
So far I've rewritten my "psio" tool to use DTrace (now it is more
robust and can access more details) and an iosnoop.d tool.
...
```

Brendan went on to an exhaustive list of what he liked and didn't like in DTrace. As one of our first major users outside of Sun, this feedback was tremendously valuable to us and very much shaped the evolution of DTrace.

And Brendan became not only one of the earliest users and foremost experts on DTrace but also a key contributor: Brendan's collection of scripts—the DTraceToolkit—became an essential factor in DTrace's adoption (and may well be how you yourself came to learn about DTrace). Indeed, one of the DTraceToolkit scripts, `shellsnoop`, remains a personal favorite of mine: It uses the syscall provider to display the contents of every read and write executed by a shell. In the early days of DTrace, whenever anyone asked whether there were security implications to running DTrace, I used to love to demo this bad boy; there's nothing like seeing someone else's password come across in clear text to wake up an audience!

Given not only Brendan's essential role in DTrace but also his gift for clearly explaining complicated systems, it is entirely fitting that he is the author of the volume now in your hands. And given the degree to which proficient use of DTrace requires mastery not only of DTrace itself but of the larger system around it, it is further appropriate that Brendan teamed up with Jim Mauro of *Solaris Internals* (McDougall and Mauro, 2006) fame. Together, Brendan and Jim are bringing you not just a book about DTrace but a book about using it in the wild, on real problems and actual systems. That is, this book isn't about dazzling you with what DTrace can do; it is about getting you closer to that moment when it is *your* butt that DTrace saves. So, enjoy the book, and remember: DTrace is a workhorse, not a show horse. Don't just read this book; put it to work and *use* it!

—Bryan Cantrill
Piedmont, California

This page intentionally left blank

Preface

"[expletive deleted] it's like they saw inside my head and gave me The One True Tool."
—A Slashdotter, in a post referring to DTrace

"With DTrace, I can walk into a room of hardened technologists and get them giggling."
—Bryan Cantrill, father of DTrace

Welcome to Oracle Solaris Dynamic Tracing—DTrace! It's been more than five years since DTrace made its first appearance in Solaris 10 3/05, and it has been just amazing to see how it has completely changed the rules of understanding systems and the applications they run. The DTrace technical community continues to grow, embracing the technology, pushing DTrace in every possible direction, and sharing new and innovative methods for using DTrace to diagnose myriad system and application problems. Our personal experience with DTrace has been an adventure in learning, helping customers solve problems faster, and improving our internal engineering efforts to analyze systems and find ways to make our technology better and faster.

The opening quotes illustrate just some of the reactions we have seen when users experience how DTrace empowers them to observe, analyze, debug, and understand their systems and workloads. The community acceptance and adoption of DTrace has been enormously gratifying to watch and participate in. We have seen DTrace ported to other operating systems: Mac OS X and FreeBSD both ship with DTrace. We see tools emerging that leverage the power of DTrace, most of which are being developed by community members. And of course feedback and comments from users over the years have driven continued refinements and new features in DTrace.

About This Book

This book is all about DTrace, with the emphasis on *using* DTrace to understand, observe, and diagnose systems and applications. A deep understanding of the details of how DTrace works is not necessary to using DTrace to diagnose and solve problems; thus, the book covers using DTrace on systems and applications, with command-line examples and a great many D scripts. Depending on your level of experience, we intend the book's organization to facilitate its use as a reference guide, allowing you to refer to specific chapters when diagnosing a particular area of the system or application.

This is not a generic performance and tools book. That is, many tools are available for doing performance analysis, observing the system and applications, debugging, and tuning. These tools exist in various places—bundled with the operating system, part of the application development environment, downloadable tools, and so on. It is probable that other tools and utilities will be part of your efforts involving DTrace (for example, using system stat tools to get a big-picture view of system resource utilization). Throughout this book, you'll see examples of some of these tools being used as they apply to the subject at hand and aid in highlighting a specific point, and coverage of the utility will include only what is necessary for clarity.

Our approach in writing this book was that DTrace is best learned by example. This approach has several benefits. The volume of DTrace scripts and one-liners included in the text gives readers a chance to begin making effective and practical use of DTrace immediately. The examples and scripts in the book were inspired by the DTraceToolkit scripts, originally created by Brendan Gregg to meet his own needs and experiences analyzing system problems. The scripts in this book encapsulate those experiences but also introduce analysis of different topics in a focused and easy-to-follow manner, to aid learning. They generate answers to real and useful questions and serve as a starting point for building more complex scripts. Rather than an arbitrary collection of programs intended to highlight a potentially interesting feature of DTrace or the underlying system, the scripts and one-liners are all based on practical requirements, providing insight about the system under observation. Explanations are provided throughout that discuss the DTrace used, as well as the output generated.

DTrace was first introduced in Oracle Solaris 10 3/05 (the first release of Solaris 10) in March 2005. It is available in all Solaris 10 releases, as well as OpenSolaris, and has been ported to Mac OS X 10.5 (Leopard) and FreeBSD 7.1. Although much of DTrace is operating system–agnostic, there are differences, such as newer DTrace features that are not yet available everywhere.[1] Using DTrace to trace operating system–specific functions, especially unstable interfaces within the kernel, will of course be very different across the different operating systems (although the same methodologies will be applicable to all). These differences are discussed throughout the book as appropriate. The focus of the book is Oracle Solaris, with key DTrace scripts provided for Mac OS X and FreeBSD. Readers on those operating systems are encouraged to examine the Solaris-specific examples, which demonstrate principles of using DTrace and often only require minor changes to execute elsewhere. Scripts that have been ported to these other operating systems will be available on the *DTrace* book Web site, *www.dtracebook.com*.

How This Book Is Structured

This book is organized in three parts, each combining a logical group of chapters related to a specific area of DTrace or subject matter.

Part I, Introduction, is introductory text, providing an overview of DTrace and its features in Chapter 1, Introduction to DTrace, and a quick tour of the D Language in Chapter 2, D Language. The information contained in these chapters is intended to support the material in the remaining chapters but does not necessarily replace the more detailed language reference available in the online, wiki-based DTrace documentation (see "Supplemental Material and References").

Part II, Using DTrace, gets you started using DTrace hands-on. Chapter 3, System View, provides an introduction to the general topic of system performance, observability, and debugging—the art of system forensics. Old hands and those who have read McDougall, Mauro, and Gregg (2006) may choose to pass over this chapter, but a holistic view of system and software behavior is as necessary to effective use of DTrace as knowledge of the language syntax. The next several chapters deal with functional areas of the operating system in detail: the I/O path—Chapter 4, Disk I/O, and Chapter 5, File Systems—is followed by Chapter 6, Network Lower-Level Protocols, and Chapter 7, Application-Level Protocols, on the network protocols. A change of direction occurs at Chapter 8, Languages, where application-level concerns become the focus. Chapter 8 itself covers programming

1. This will improve after publication of this book, because other operating systems include the newer features.

languages and DTrace's role in the development process. Chapter 9, Applications, deals with the analysis of applications. Databases are dealt with specifically in Chapter 10, Databases.

Part III, Additional User Topics, continues the "using DTrace" theme, covering using DTrace in a security context (Chapter 11, Security), analyzing the kernel (Chapter 12, Kernel), tools built on top of DTrace (Chapter 13, Tools), and some tips and tricks for all users (Chapter 14, Tips and Tricks).

Each chapter follows a broadly similar format of discussion, strategy suggestions, checklists, and example programs. Functional diagrams are also included in the book to guide the reader to use DTrace effectively and quickly.

For further sources of information, see the online "Supplemental Material and References" section, as well as the annotated bibliography of textbook and online material provided at the end of the book.

Intended Audience

DTrace was designed for use by technical staff across a variety of different roles, skills, experience, and knowledge levels. That said, it is a software analysis and debugging tool, and any substantial use requires writing scripts in D. D is a structured language very similar to C, and users of that language can quickly take advantage of that familiarity. It is assumed that the reader will have some knowledge of operating system and software concepts and some programming background in scripting languages (Perl, shell, and so on) and/or languages (C, C++, and so on).

In addition, you should be familiar with the architecture of the platform you're using DTrace on. Textbooks on Solaris, FreeBSD, and Mac OS X are detailed in the bibliography.

To minimize the level of programming skill required, we have provided many DTrace scripts that you can use immediately without needing to write code. These also help you learn how to write your own DTrace scripts, by providing example solutions that are also starting points for customization. The DTraceToolkit[2] is a popular collection of such DTrace scripts that has been serving this role to date, created and mostly written by the primary author of this book. Building upon that success, we have created a book that is (we hope) the most comprehensive source for DTrace script examples.[3]

---

2. This is linked on *www.brendangregg.com/dtrace.html* and *www.dtracebook.com*.
3. The DTraceToolkit now needs updating to catch up!

This book will serve as a valuable reference for anyone who has an interest in or need to use DTrace, whether it is a necessary part of your day job, a student studying operating systems, or a casual user interested in figuring out why the hard drive on your personal computer is clattering away doing disk I/Os.

Specific audiences for this book include the following.

Systems administrators, database administrators, performance analysts, and support staff responsible for the care and feeding of their production systems can use this book as a guide to diagnose performance and pathological behavior problems, understand capacity and resource usage, and work with developers and software providers to troubleshoot application issues and optimize system performance.

Application developers can use DTrace for debugging applications and utilizing DTrace's User Statically Defined Tracing (USDT) for inserting DTrace probes into their code.

Kernel developers can use DTrace for debugging kernel modules.

Students studying operating systems and application software can use DTrace because the observability that it provides makes it a perfect tool to supplement the learning process. Also, there's the implementation of DTrace itself. DTrace is among the most well-thought-out and well-designed software systems ever created, incorporating brilliantly crafted solutions to the extremely complex problems inherent in building a dynamic instrumentation framework. Studying the DTrace design and source code serves as a world-class example of software engineering and computer science.

Note that there is a minimum knowledge level assumed on the part of the reader for the topics covered, allowing this book to focus on the application of DTrace for those topics.

Supplemental Material and References

Readers are encouraged to visit the Web site for this book: *www.dtracebook.com*.

All the scripts contained in the book, as well as reader feedback and comments, book errata, and subsequent material that didn't make the publication deadline, can be downloaded from the site.

Brendan Gregg's DTraceToolkit is free to download and contains more than 200 scripts covering every everything from disks and networks to languages and the kernel. Some of these are used in this text: *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit*.

The DTrace online documentation should be referenced as needed: *http://wikis.sun.com/display/DTrace/Documentation*.

The OpenSolaris DTrace Community site contains links and information, including projects and additional sources for scripts: *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/*.

The following texts (found in the bibliography) can be referenced to supplement DTrace analysis and used as learning tools:

McDougall and Mauro, 2006

McDougall, Mauro, and Gregg, 2006

Gove, 2007

Singh, 2006

Neville-Neil and McKusick, 2004

Acknowledgments

The authors owe a huge debt of gratitude to Deirdré Straughan for her dedication and support. Deirdré spent countless hours reviewing and editing material, substantially improving the quality of the book. Deirdré has also dedicated much of her time and energy to marketing and raising awareness of DTrace and this book both inside and outside of Oracle.

Dominic Kay was tireless in his dedication to careful review of every chapter in this book, providing detailed commentary and feedback that improved the final text tremendously. Darryl Gove also provided extraordinary feedback, understanding the material very well and providing numerous ideas for improving how topics are explained. And Peter Memishian provided incredible feedback and expertise in the short time available to pick through the longest chapter in the book, Chapter 6, and greatly improve its accuracy.

Kim Wimpsett, our copy editor, worked through the manuscript with incredible detail and in great time. With so many code examples, technical terms, and output samples, this is a very difficult and tricky text to edit. Thanks so much for the hard work and patience.

We are very grateful to everyone who provided feedback and content on some or all of the chapters in the short time frame available for such a large book, notably, Alan Hargreaves, Alan Maguire, Andrew Krasny, Andy Bowers, Ann Rice, Boyd Adamson, Darren Moffatt, Glenn Brunette, Greg Price, Jarod Jenson, Jim Fiori, Joel Buckley, Marty Itzkowitz, Nasser Nouri, Rich Burridge, Robert Watson, Rui Paulo, and Vijay Tatkar.

A special thanks to Alan Hargreaves for his insights and comments and contributing his USDT example and case study in Appendix E.

Thanks to Chad Mynhier and Tariq Magdon-Ismail for their contributions.

Thanks to Richard McDougall for so many years of friendship and inspiration and for the use of the RMCplex.

We'd like to thank the software engineers who made this all possible in the first place, starting with team DTrace at Sun Microsystems (Bryan Cantrill, Mike Shapiro, and Adam Leventhal) for inventing DTrace and developing the code, and team DTrace at Apple for their port of not only DTrace but many DTraceToolkit scripts (Steve Peters, James McIlree, Terry Lambert, Tom Duffy, and Sean Callanan); and we are grateful for the work that John Birrell performed to port DTrace to FreeBSD. We'd also like to thank the software engineers, too numerous to mention here, who created all the DTrace providers we have demonstrated throughout the book.

Thanks to the worldwide community that has embraced DTrace and generated a whirlwind of activity on the public forums, such as dtrace-discuss. These have been the source of many great ideas, examples, use cases, questions, and answers over the years that educate the community and drive improvements in DTrace.

And a special thanks to Greg Doench, senior editor at Pearson, for his help, patience, and enthusiasm for this project and for working tirelessly once all the material was (finally) delivered.

Personal Acknowledgments from Brendan

Working on this book has been an enormous privilege, providing me the opportunity to take an amazing technology and to demonstrate its use in a variety of new areas. This was something I enjoyed doing with the DTraceToolkit, and here was an opportunity to go much further, demonstrating key uses of DTrace in more than 50 different topics. This was also an ambitious goal: Of the 230+ scripts in this book, only 45 are from the DTraceToolkit; most of the rest had to be newly created and are released here for the first time. Creating these new scripts required extensive research, configuration of application environments and client workloads, experimentation, and testing. It has been exhausting at times, but it is satisfying to know that this should be a valuable resource for many.

A special thanks to Jim for creating the DTrace book project, encouraging me to participate, and then working hard together to make sure it reached completion. Jim is an inspiration to excellence; he co-authored *Solaris Internals* (McDougall and Mauro, 2006) with Richard McDougall, which I studied from cover to cover while I was learning DTrace. I was profoundly impressed by its comprehensive coverage, detailed explanations, and technical depth. I was therefore honored to be invited to collaborate on this book and to work with someone who had the experience and desire to take on a similarly ambitious project. Jim has an amazing can-do attitude and willingness to take on hard problems, which proved essential as we worked through the numerous topics in this book. Jim, thanks; we somehow survived!

Thanks, of course, are also due to team DTrace; it's been a privilege to work with them and learn from them as part of the Fishworks team. Especially sitting next to Bryan for four years: Learning from him, I've greatly improved my software analysis skills and will never forget to separate problems of implementation from problems of abstraction.

Thanks to the various Sun/Oracle teams I regularly work with, share problems with, and learn from, including the Fishworks, Performance Availability Engineering (PAE), Independent Software Vendor (ISV) engineering, and ZFS teams.

Thanks to Claire, for the love, support, and patience during the many months this was to take, and then the many months beyond which it actually took to complete. These months included the birth of our child, Mitchell, making it especially tough for her when I was working late nights and weekends on the book.

—Brendan Gregg
Walnut Creek, California (formerly Sydney, Australia)
September 2010

Personal Acknowledgments from Jim

Working on this book was extremely gratifying and, to a large degree, educational. I entered the project completely confident in my knowledge of DTrace and its use for observing complex systems. A few months into this project, I quickly realized I had only scratched the surface. It's been enormously rewarding to be able to improve my knowledge and skills as I worked on this book, while at the same time improving and adding more value to the quality of this text.

First and foremost, a huge thank you to Brendan. Brendan's expertise and sheer energy never ceased to amaze me. He consistently produced huge amounts of material—DTrace scripts, one-liners, and examples—at a rate that I would have never thought humanly possible. He continually supplied an endless stream of ideas, constantly improving the quality of his work and mine. He is uncompromising in his standards for correctness and quality, and this work is a reflection of Brendan's commitment to excellence. Brendan's enthusiasm is contagious—throughout this project, Brendan's desire to educate and demonstrate the power of DTrace, and its use for solving problems and understanding software, was an inspiration. His expertise in developing complex scripts that illuminate the behavior of a complex area of the kernel or an application is uncanny. Thanks, mate; it's been a heck of a ride. More than anything, this is your book.

Thanks to my manager, Fraser Gardiner, for his patience and support.

I want to thank the members of Fraser's team who I have the opportunity to work with and learn from every day: Andy Bowers, Matt Finch, Calum Mackay, Tim Uglow, and Rick Weisner, all of whom rightfully belong in the "scary smart" category.

Speaking of "scary smart," a special thanks to my friend Jon Haslam for answering a constant stream of DTrace questions and for his amazing contributions to DTrace.

Thanks to Chad Mynhier for his ideas, contributions, patience, and understanding.

Thanks to my friends Richard McDougall and Bob Sneed for all the support, advice, and time spent keeping me going over the years. And a special thank-you to Richard for use of the RMCplex.

Thanks to Donna, Frank, and Dominic for their love, patience, and support.

Thanks Lisa, for the love, support, and inspiration and for just being you.

—Jim Mauro
Green Brook, New Jersey
September 2010

About the Authors

Brendan Gregg is a performance specialist at Joyent and is known worldwide in the field of DTrace. Brendan created and developed the DTraceToolkit and is the coauthor of *Solaris Performance and Tools* (McDougall, Mauro, and Gregg, 2006) as well as numerous articles about DTrace. He was previously the performance lead for the Sun/Oracle ZFS storage appliance and a software developer on the Fishworks advanced development team at Sun, where he worked with the three creators of DTrace. He has also worked as a system administrator, performance consultant, and instructor, and he has taught DTrace worldwide including workshops that he authored. His software achievements include creating the DTrace IP, TCP, and UDP providers; the DTrace JavaScript provider; and the ZFS L2ARC. Many of Brendan's DTrace scripts are shipped by default in Mac OS X.

Jim Mauro is a senior software engineer for Oracle Corporation. Jim works in the Systems group, with a primary focus on systems performance. Jim's work includes internal performance-related projects, as well as working with Oracle customers on diagnosing performance issues on production systems. Jim has 30 years of experience in the computer industry, including 19 years with Sun Microsystems prior to the acquisition by Oracle. Jim has used DTrace extensively for his performance work since it was first introduced in Solaris 10 and has taught Solaris performance analysis and DTrace for many years.

Jim coauthored the first and second editions of *Solaris Internals* (McDougall and Mauro, 2006) and *Solaris Performance and Tools* (McDougall, Mauro, and Gregg, 2006) and has written numerous articles and white papers on various aspects of Solaris performance and internals.

This page intentionally left blank

1

Introduction to DTrace

This chapter introduces you to DTrace.

What Is DTrace?

DTrace[1] is an observability technology that allows you to answer countless questions about how systems and applications are behaving in development and in production. DTrace *empowers* users in ways not previously possible by enabling the dynamic instrumentation of unmodified kernel and user software.

Created by Bryan Cantrill, Mike Shapiro, and Adam Leventhal, DTrace was first introduced in Solaris 10 3/05 (the first release of Solaris 10) in March 2005. It is now available in all Solaris 10 releases, as well as OpenSolaris, Mac OS X beginning with release 10.5 (Leopard), and FreeBSD beginning with release 7.1.

Why Do You Need It?

Understanding what is going on in a software system has been a challenge for as long as such systems have existed. Tools and instrumentation frameworks were already available, such as language-specific debuggers and profiling tools, operat-

1. DTrace is short for Oracle Solaris Dynamic Tracing Facility.

ing system–specific utilities built on precompiled instrumentation points, and so on. But these tools suffered drawbacks: They added a performance burden to the running system, required special recompiled versions of the software to function, needed several different tools to give a complete view of system behavior, limited available instrumentation points and data, and required significant postprocessing to create meaningful information from the gathered data.

DTrace solves all these problems, but it also does much more; it revolutionizes software instrumentation. It is so powerful that we're still learning the full extent of its potential uses, extending its capabilities with new features and functionality, and devising new and innovative ways to leverage the power and flexibility it brings.

Capabilities

DTrace's broad range of capabilities make it useful for troubleshooting any software function, including entry arguments and return values. This can be done in production, without restarting or modifying applications or operating systems. You can make detailed observations of devices, such as disks or network interfaces, and explore the use of core resources such as CPU and memory. DTrace gives you insight into where the kernel is spending time and how applications are functioning. It is particularly useful in performance analysis and capacity-planning tasks such as finding latencies and understanding how resources are being used.

Figure 1-1 shows the software stack components found in a typical production workload. The number of applications, languages, and so on, available today is enormous, as is the number of tools and methods available for system analysis.

Given this problem space, it is interesting to compare the comprehensive coverage DTrace provides to the cohort of other available analysis tools. Consider the tool sets required to examine every layer in Figure 1-1, as shown in Table 1-1 for Oracle Solaris.

Different layers of the software stack typically require different tools and utilities for analysis and debugging, none of which provides a complete system view. The bundled system tools fall into one of several categories:

Process/thread centric: Examining process statistics from tools including `prstat(1)`, `ps(1)`, and `top(1)`

System resource centric: System tools to examine resource usage by component, including `vmstat(1)`, `mpstat(1)`, and `iostat(1)`

Traditional debuggers: Used to inspect the execution of code, such as `dbx`, `mdb(1)`, and Oracle Sun Studio

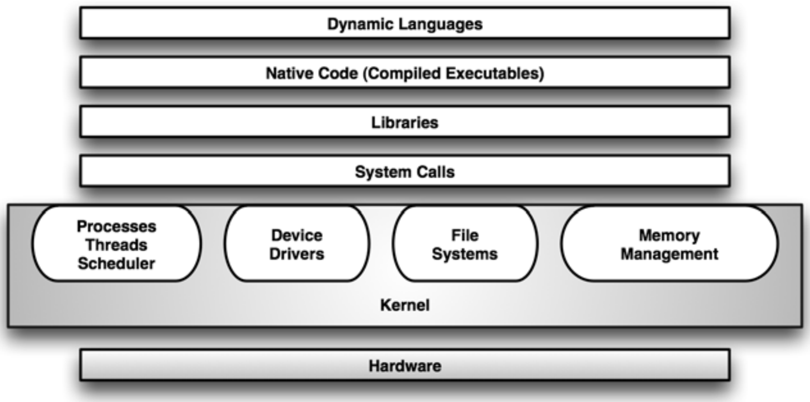


Figure 1-1 The software stack

Many of these tools can provide useful starting points for analysis, which can then be explored in depth with DTrace. Or, you can use DTrace from the outset to examine the entire software stack from one consistent tool.

Although it is a system-level feature, DTrace is useful well beyond the operating system. It provides the application programmer with observability across the entire OS and application stack, giving insight into the data-path traffic and network activity generated by applications, as well as the internal runtime behaviors of applications themselves. It can be used both to step through execution logic and to profile behavior in a statistical manner.

Table 1-1 Software Stack Tools

Layer	Layer Examples	Previous Analysis	DTrace Visibility
Dynamic languages	Java, Ruby, PHP, and so on	Debuggers	Yes, with providers
Native code	Compiled C/C++ code	Debuggers, `truss`	Yes
Libraries	`/usr/lib/*`, compiled code	`apptrace`, `sotruss`, truss	Yes
System calls	`man -s 2`, `read(2)`, and so on	`truss`	Yes
Kernel	Proc/threads, FS, VM, and so on	`prex`; `tnf`, `lockstat`, `mdb`, `adb`	Yes
Hardware	Disk HBA, NIC, CPU, and so on	`cpustat`, `kstats`, and so on	Indirectly, yes

Dynamic and Static Probes

Previous tracing tools used static instrumentation, which adversely affects performance even when not enabled. DTrace supports both *static* and *dynamic* instrumentation. That is, the DTrace framework is designed to enable and disable instrumentation points in unmodified software dynamically, on the fly, while the system and applications are running. DTrace also provides a facility for developers to insert custom instrumentation points in their code (static tracing).

DTrace can insert instrumentation points called *probes* dynamically into running software. A probe can trace execution flow through code, collecting relevant data along the way. When a probe has been enabled and the code where the probe has been inserted executes, the probe will *fire*, showing that it hit the instrumented probe point in the code flow.

DTrace supports static probes by including instruction no-operations (NOPs) at the desired probe points in compiled software, which become the real instructions when in use. The disabled probe effect because of the addition of NOPs is near-zero.

What happens when the probe fires is entirely up to you. You can collect and aggregate data, take time stamps, collect stack traces, and so on. These choices will be explored extensively throughout this book. Once the desired actions have been taken, the code resumes executing normally: The software behaves just as if the probe were not present. Dynamically generated probes alter code only when they are in use; when disabled, their effect on performance is zero.

Among the many benefits of DTrace's innovative design are CPU and memory utilization—the framework makes minimal demand on CPU cycles and memory. DTrace includes a logical predicate mechanism that allows actions to be taken only when user-specified conditions are met, pruning unwanted data *at the source*. DTrace thus avoids retaining, copying, and storing data that will ultimately be discarded.

DTrace Features

DTrace features[2] include the following.

> **Dynamic instrumentation**: Performance will always take a hit with static instrumentation, even when probes are disabled. To achieve the zero probe effect required for production systems, DTrace can also use dynamic instrumentation. When DTrace is not in use, there is absolutely no effect on system performance.

2. This feature list is from *Dynamic Instrumentation of Production Systems* (Cantrill, Shapiro, and Leventhal, 2005).

Unified instrumentation: Beyond requiring different tools for different aspects of the operating system, earlier approaches also required different tools for the operating system vs. applications. DTrace can dynamically instrument both user- *and* kernel-level software and can do so in a *unified* manner whereby both data and control flow can be followed across the user/kernel boundary.

Arbitrary-context kernel instrumentation: DTrace can instrument virtually all of the kernel, including delicate subsystems such as the scheduler and synchronization facilities.

Data integrity: DTrace reports any errors that prevent trace data from being recorded. If there are no errors, DTrace guarantees data integrity; recorded data cannot be silently corrupted or lost.

Arbitrary actions: Because it is dynamic, the actions taken by DTrace at any given point of instrumentation are not defined or limited *a priori*. You can enable any probe with an arbitrary set of actions.

Safety: DTrace guarantees absolute safety of user-defined actions: Runtime errors such as illegal memory accesses are caught and reported. Indeed, safety was central to the design of DTrace from its inception, given that the target environment for using DTrace are production systems.[3] In addition to runtime checking of user-defined actions, DTrace includes a watchdog timer, verifying that the target system is reasonably alive and responsive, and includes other safety mechanisms.

Predicates: A logical predicate mechanism allows actions to be taken only when user-specified conditions are met. Unwanted data is discarded *at the source*—never retained, copied, or stored.

A high-level control language: DTrace is equipped with an expressive C-like scripting language known as *D*. It supports all ANSI C operators, which may be familiar to you and reduce your learning curve, and allows access to the kernel's variables and native types. D offers user-defined variables, including global variables, thread-local variables, and associative arrays, and it supports pointer dereferencing. This, coupled with the runtime safety mechanisms of DTrace, means that structure chains can be safely traversed in a predicate or action.

A scalable mechanism for aggregating data: Data retention can be further minimized by statistical aggregation. This coalesces data as it is generated, reducing the amount that percolates through the framework by a factor

3. Of course, DTrace can be used across the entire system's spectrum—development, QA, test, and so on.

of the number of data points. So, instead of handing a large quantity of data to user-land software for summarization, DTrace can perform certain summaries in the kernel.

Speculative tracing: DTrace has a mechanism for speculatively tracing data, deferring the decision to commit or discard the data to a later time. This eliminates the need for most postprocessing when exploring sporadic aberrant behavior, such as intermittent error events.

Heterogeneous instrumentation: Where tracing frameworks have historically been designed around a single instrumentation methodology, DTrace is extensible to new instrumentation problems and their solutions. In DTrace, the instrumentation providers are formally separated from the probe processing framework by a well-defined API, allowing fresh dynamic instrumentation technologies to plug in to and exploit the common framework.

Scalable architecture: DTrace allows for many tens of thousands of instrumentation points (even the smallest systems typically have on the order of 30,000 such points) and provides primitives for subsets of probes to be efficiently selected and enabled.

Virtualized consumers: Everything about DTrace is virtualized per consumer: Multiple consumers can enable the same probe in different ways, and a single consumer can enable a single probe in different ways. There is no limit on the number of concurrent DTrace consumers.

Privileges: DTrace is secure. By default, only the root user (system administrator) has the privileges required to use DTrace. In Solaris, the Process Rights facility can be configured to allow DTrace to be used by nonroot users. This is covered in more detail in Chapter 11, Security.

In this chapter, we provide a jump-start into DTrace, with example one-liners and enough coverage of the underlying architecture and terminology to get you going.

A First Look

DTrace has been described as a tool that "allows you to ask arbitrary questions about what the system is doing, and get answers."[4] This section provides examples of DTrace fulfilling that promise and demonstrating its expressive power.

4. This often-used phrase to describe DTrace was first used by Bryan Cantrill, the coinventor of DTrace.

Consider getting beyond basic disk I/O statistics that provide reads and writes per second on a per-device basis (such as `iostat(1M)` output) to something much more meaningful. How about knowing which *files* are being read and which *processes* are reading them? Such information is extremely helpful in understanding your application and workload but near impossible to get on most operating systems. With DTrace, however, it is trivially easy. Here it is traced at the file system level so that all I/O can be seen:

```
opensolaris# dtrace -n 'syscall::read:entry /execname != "dtrace"/ {
      @reads[execname, fds[arg0].fi_pathname] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C
bash            /proc/1709/psinfo                                             1
loader          /zp/space/f2                                                  1
nscd            /etc/user_attr                                                1
bash            /export/home/mauroj/.bash_history                             2
loader          /zp/space/f3                                                  2
nscd            /etc/group                                                    2
su              /etc/default/su                                               8
su              /devices/pseudo/sy@0:tty                                      9
bash            /dev/pts/5                                                   66
Xorg            /devices/pseudo/conskbd@0:kbd                               152
gnome-terminal  /devices/pseudo/clone@0:ptm                                 254

Script read-syscall.d
```

We use the DTrace command (`dtrace(1M)`) to enable a probe at the entry point of the `read(2)` system call. A filter, in / /, is used to skip tracing system calls by `dtrace` itself. The action taken, in { }, counts the number of reads by process name and path name, derived using DTrace variables.

`dtrace(1M)` reported that we matched one probe, and the DTrace kernel subsystem gathered the requested data until the command was terminated using Ctrl-C. The output shows the process name, filename, and number of reads, in ascending order.

You can see that we are able to observe a typical workload component (file system I/O) in a way that has real meaning to us in terms of the running application (processes and filenames), by running a relatively short DTrace command.

As another example, here we use DTrace to observe what happens when a very common system command, `man(1)`, is executed on Solaris. In this example, we use `man ls`.

```
opensolaris# dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
dtrace: description 'exec-success ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  24953        exec_common:exec-success   man ls
  0  24953        exec_common:exec-success   neqn /usr/share/lib/pub/eqnchar -
  0  24953        exec_common:exec-success   col -x
```

continues

```
   0  24953         exec_common:exec-success   sh -c less -siM /tmp/mp1RaGim
   0  24953         exec_common:exec-success   less -siM /tmp/mp1RaGim
   1  24953         exec_common:exec-success   sh -c cd /usr/man;
tbl /usr/man/man1/ls.1 |neqn /usr/share/lib/pub/eqnchar - |n
   1  24953         exec_common:exec-success   tbl /usr/man/man1/ls.1
   1  24953         exec_common:exec-success   nroff -u0 -Tlp -man -
   1  24953         exec_common:exec-success   sh -c trap '' 1 15;
/usr/bin/mv -f /tmp/mp1RaGim /usr/man/cat1/ls.1 2>
/dev/nul
  1  24953         exec_common:exec-success   /usr/bin/mv -f
/tmp/mp1RaGim /usr/man/cat1/ls.1

Script chpt1_exec.d
```

The DTrace probe enabled is `proc:::exec-success`[5], a probe that fires when one of the `exec(2)` family of system calls successfully loads a new process image—a normal part of process creation. The user-defined action when the probe fires is to execute the DTrace `trace()` function to print the argument list of the current process, if available.[6] The first four columns of output (starting at the left) consist of the information DTrace provides by default whenever a probe fires. We see the CPU the probe fired on, the probe ID, and part of the probe name. The last column is the output generated from our `trace()` function, which is the argument list of the process.

Starting at the top line of the output, we see `man ls`, followed by the typical series of exec'd commands executed to format and display a man page (`neqn`, `col`, `sh`, `less`, `sh`, `tbl`, `nroff`, `sh`, and lastly `mv`). Here again we see how the observability that DTrace probes provide, which allows us to understand all aspects of the work our systems actually do, whether we're looking at the execution of a common command or getting a systemwide view of disk I/O activity. Consider how difficult it would be to do this with earlier tools!

Overview

In this section, we provide an overview of the various components that make up DTrace. Table 1-2 is a glossary of key DTrace terms; there is also a full glossary toward the end of this book.

5. On FreeBSD, this probe was `proc:::exec_success` and is now being updated to `proc:::exec-success`.
6. The full argument list is not currently shown on Mac OS X and FreeBSD at the time of writing this book. (If you are a developer and would like to help fix this, the starting point is to `grep` for `pr_psargs` in `/usr/lib/dtrace/darwin.d` for Mac OS X, and `/usr/lib/dtrace/psinfo.d` on FreeBSD; they need to translate the `arg` string from the kernel.)

Table 1-2 DTrace Terms

Term	Definition
D language	This is the defined set of terms, syntax, semantics, and functions for using DTrace. Note that using DTrace either from the command line or by running a script equates to the execution of a D program written in the D language.
Consumer	This is a user-mode program that calls into the underlying DTrace framework.
Provider	Part of the DTrace framework, providers manage probes associated with a specific subsystem.
Probe	This is a user-enabled point of instrumentation.
Predicate	This is a user-defined conditional statement evaluated (if present) when probes fire that enables data capture only if a specific condition or set of conditions is true.
Clause	This is the user-defined actions to take when a probe fires.
Variable	As with other programming languages, a variable in DTrace provides storage for a particular type of data object. DTrace supports user-defined variables, as well as a rich set of built-in variables.
Aggregation	This is a variable type and set of related functions that provide for data coalescing and representation.
Function	This is any one of many DTrace functions that can be called as part of a user-defined action in D.

Consumers

There are currently four bundled commands in Solaris categorized as DTrace consumers, meaning they utilize the DTrace framework by calling into the routines in the DTrace library. `dtrace(1M)` is the general-use DTrace consumer, allowing for enabling probes and specifying predicates and actions to take when probes fire. `lockstat(1M)` is a utility for collecting statistics on kernel locks (mutual exclusion—or *mutex*—locks and reader/writer locks) and for generating kernel profiles.[7] `plockstat(1M)` provides statistics on user-level mutex locks and reader/writer locks. `intrstat(1M)` provides statistics on device interrupts.

7. `lockstat(1M)` has been available in Solaris since Solaris 7; in Solaris 10, it was modified to use the DTrace framework. It is functionally identical to `lockstat(1M)` in pre–Solaris 10 releases.

Probes

A *probe* is a point of instrumentation, typically a specific location in program flow, although some probes are time-based, as we will discuss later. To list all the probes available for your use, simply use `dtrace(1M)` with the `-l` flag.

```
solaris# dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
[...]
  972        fbt          physmem                 physmem_map_addrs entry
  973        fbt          physmem                 physmem_map_addrs return
  974        fbt          physmem                   physmem_getpage entry
 2884       proc          genunix                         proc_exit exit
 2885       proc          genunix                          lwp_exit lwp-exit
 2886       proc          genunix                         proc_exit lwp-exit
 2887       proc          genunix                       exec_common exec-success
 2888       proc          genunix                       exec_common exec-failure
 2889       proc          genunix                       exec_common exec
 2890    sysinfo          genunix                       exec_common sysexec
 2891   sysevent          genunix                    queue_sysevent post
 2892   sysevent          genunix                    evch_chpublish post
 2893        sdt          genunix                     netstack_hold netstack-inc-ref
[...]
```

The listing shows the probe identifier (ID) for internal use by DTrace, followed by a probe name of four components. As an introduction to probe terminology (this is covered again in Chapter 2, D Language), probe names are specified using the following:

```
provider:module:function:name
```

where

`provider`: Providers are libraries of probes that instrument a specific area of the system (for example, sched) or a mode of tracing (for example, `fbt`). New providers are written over time and added to newer releases (for example, ip, tcp, perl, python, mysql, and so on).

`module`: This is the kernel module where the probe is located. For user-land probes, it reflects the shared object library that contains the probe.

`function`: This is the software function that contains this probe.

`name`: This is a meaningful name to describe the probe. For example, names such as `entry` and `return` are probes that fire at the entry and return of the corresponding function.

The number of probes available on an operating system will vary based on loaded kernel modules and available providers for that version. To illustrate this, we list in the following example probes and counted lines of output on the different operating systems. The number reported is the number of currently available probes plus a header line.

```
Solaris 10 10/08
solaris10# dtrace -l | wc -l
   73742

Mac OS X 10.5.6
macosx# dtrace -l | wc -l
   23378

OpenSolaris 2008.11
opensolaris# dtrace -l | wc -l
   55665

FreeBSD 7.1
freebsd# dtrace -l | wc -l
   33207
```

When enabling dynamically generated probes, these counts can become much larger (hundreds of thousands of probes).

Providers

Providers are libraries of probes. Most exist to provide logical abstractions of complex areas of the system, providing probes with intuitive names and providing useful data related to that probe. They allow you to instrument software without necessarily needing to study source code or become an expert in an area targeted for instrumentation.

The core providers available in Solaris 10, OpenSolaris, Mac OS X, and FreeBSD are as follows:

dtrace: The dtrace provider manages housekeeping probes to define what happens when a script `BEGIN`s, `END`s, or `ERROR`s.

syscall: The syscall provider manages probes at the entry and return points for all available system calls—the API by which applications request the services of the operating system.

proc: The proc provider manages probes specific to process- and thread-related events.

profile: The profile provider manages probes used for time-based data collection.

fbt: The fbt provider manages function boundary tracing, managing probes at the entry and exit points of almost all kernel functions.

lockstat: The lockstat provider manages probes that cover the operation of kernel synchronization primitives.

The proc provider is an example of static instrumentation, because these probe points have been chosen, instrumented, and built into the kernel. The fbt provider is an example of dynamic instrumentation; the probes it provides are generated dynamically from the current kernel version.

Many other providers may or may not be available on your version of operating system kernel and application software; they are discussed in later chapters of this book. These include the io provider for disk and back-end device I/O, available on Solaris 10, OpenSolaris, and Mac OS X. The io provider is a good example of the role of providers, because it provides user-friendly probes for tracing disk I/O without the user needing to learn and instrument kernel internals. Listing the io provider probes on Mac OS X, for example, is done using `-l` to list probes and `-P` to specify a provider name:

```
macosx# dtrace -l -P io
   ID   PROVIDER            MODULE                          FUNCTION NAME
18501         io       mach_kernel                      buf_strategy start
18514         io       mach_kernel                       buf_biodone done
18516         io       mach_kernel              buf_biowait_callback wait-done
18517         io       mach_kernel              buf_biowait_callback wait-start
18518         io       mach_kernel                       buf_biowait wait-done
18519         io       mach_kernel                       buf_biowait wait-start
```

The io provider gives us a small number of probes with intuitive names. These names are listed in the `NAME` column: `start` fires when a disk I/O request is started, and `done` fires when a disk I/O request is completed. Information about these events is made available via argument variables, which include the size and offset of the disk I/O. The io provider is described fully in Chapter 4, Disk I/O.

The reference for all DTrace providers is the DTrace Guide,[8] which lists the probes and arguments that each make available. Various providers are also demonstrated throughout this book, and Appendix C, Provider Arguments, lists provider probes and arguments.

8. This is currently available at *http://wikis.sun.com/display/DTrace/Documentation*.

Predicates

DTrace provides a facility for collecting data only when a user-defined condition or set of conditions is true. For example, a specific process name can be targeted for data collection or when you're interested only in disk reads (not writes), network transmits (not receives), specific error conditions, and so on. A predicate is an optional conditional statement that is evaluated after its associated probes fire. If the conditions evaluate true, the user-defined action is taken. If the predicate evaluates false, no action is taken.

Actions

The action we refer to here is the body of the D program, following the probe description and optional predicate, where the user defines what to do when a probe fires. These actions are defined within a probe's *clause*. Actions may include collecting data, capturing time stamps, gathering stack traces, and so on. It is entirely up to you to determine what happens when a probe fires, and if present, a predicate evaluates true.

Aggregations

DTrace provides the ability to coalesce data at the point of collection using a predefined set of aggregating functions and storing the results of those functions in a special DTrace variable called an *aggregation*. Aggregations minimize the amount of data returned to the consumer and enable presenting the data in an immediately useful format; no postprocessing is required before analysis can begin.

For example, here we use an aggregation to examine disk I/O size as a distribution plot:

```
macosx# dtrace -n 'io:::start { @bytes = quantize(args[0]->b_bcount); }'
dtrace: description 'io:::start ' matched 1 probe
^C

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@                                  129
            1024 |                                         0
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@                       318
            8192 |@@@@@@@                                  130
           16384 |@@@@                                     63
           32768 |@@                                       35
           65536 |@                                        18
          131072 |@                                        13
          262144 |                                         4
          524288 |                                         3
         1048576 |                                         1
         2097152 |                                         0
```

Aggregation variables are prefixed with @ and are populated using aggregating functions—in this case quantize(), which summarizes data for later printing as a distribution plot. The previous output shows that the most frequent I/O size requested was between 4KB and 8KB while this one-liner was tracing.

D Language

The format of a DTrace program is consistent whether you are using DTrace from the command line or writing D scripts. This is covered in Chapter 2 and summarized here as an introduction.

There are essentially three components to a DTrace invocation:

- The probes
- An optional predicate
- An optional probe clause, containing the actions to take when the probe fires

Here is an example of using dtrace on a command line:

```
# dtrace -n 'probe /predicate/ { actions }'
```

Here are the components as they appear in a D script:

```
#!/usr/sbin/dtrace -s
probe
/predicate/
{
        actions
}
```

The probe, as described previously, defines the point of instrumentation. More than one probe can be defined (comma-separated) if the same predicate and action are desired. Alternatively, multiple probes can be defined with different predicates and actions. DTrace provides some flexibility in how you specify the probe names; every probe need not be fully qualified with each of the four fields specified. For example, you could enable a probe at the entry point of every system call using this:

```
# dtrace -n 'syscall:::entry'
dtrace: description 'syscall:::entry' matched 235 probes
CPU     ID                    FUNCTION:NAME
  0  79352                      ioctl:entry
```

```
  0  79352                       ioctl:entry
  0  79490                   sysconfig:entry
  0  79490                   sysconfig:entry
[...]
```

In the previous example code, the function field, which for the `syscall` provider is the name of the system call, is left blank in the probe name. DTrace will treat blank fields as wildcards and enable all probes matching the other fields defined in the DTrace invocation. In this example, the `entry` point of every system call was enabled. Because a probe clause was not specified, DTrace took the default action, which is to print the `CPU ID` of the CPU that executed the code (causing the probe to fire), the numeric `ID` of the probe, and the `FUNCTION` and `NAME` fields of the probe.

When included, the probe clause follows the probe name, is enclosed in curly braces, and contains the user-defined *actions* to be taken when the probe fires. Extending the previous example, we can easily modify our D program to frequency count the name of the system call and the name of the program that executed the system call, in a simple statement in the probe clause:

```
# dtrace -n 'syscall:::entry { @sc[execname, probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 235 probes
^C
  . . .
  dtgreet         pollsys                                         20
  java            stat64                                          45
  java            pollsys                                         88
  syslogd         getmsg                                          160
  syslogd         pollsys                                         160
  dtrace          p_online                                        256
  syslogd         lwp_park                                        640
  dtrace          ioctl                                           1599

Script syscall-1.d
```

This example used an aggregation to perform a frequency count, which was specified using the `count()` aggregation function. Aggregations can be indexed using *keys*. In this example, the keys were the process name (`execname`) and function field (`probefunc`, which for this provider contains the system call name); these keys are printed as columns in the output, sorted on the value.

The final structure element to discuss is the *predicate*. Predicates are conditional statements that get evaluated after the probe fires but before any actions in the clause are executed. If the expression in the predicate is evaluated as `TRUE`, the clause is entered, and the actions in the clause are executed. If the predicate evaluates `FALSE`, no action is taken when the probe fires. Predicates add great power to DTrace, giving users the ability to filter the data collected and returned, based on specific conditions of interest. For example, here we can ask DTrace to not capture data when the process executing is `dtrace(1M)` itself, by adding a simple predicate to the example:

```
# dtrace -n 'syscall:::entry /execname != "dtrace"/
{ @sc[execname, probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 235 probes
^C
  . . .
  dtgreet         pollsys                                 20
  java            stat64                                  45
  java            pollsys                                 88
  syslogd         getmsg                                  160
  syslogd         pollsys                                 160
  syslogd         lwp_park                                640
  loader          read                                    2232
Script syscall-2.d
```

We added the predicate `/execname != "dtrace"/` after the probe. The `!=` operator is a relational operator that means *not equal*. If the name of the process running on the CPU when the probe fires is not `dtrace`, the action in the clause is taken. If the name is `dtrace`, the predicate evaluates `FALSE`, and no action is taken.

DTrace supports a superset of ANSI-C operators that can be used to build complex and powerful expressions, including relational, logical, bitwise, and arithmetic operators; this is covered in Chapter 2.

Architecture

Having described the terms and use of DTrace, we will now take a brief look at how DTrace is structured. Figure 1-2 illustrates the major components of the DTrace framework.

The DTrace consumers execute in user mode and use the `libdtrace.so` library. This library is not a public interface; general-purpose use of DTrace is via the `dtrace(1M)` command, as well as `lockstat(1M)`, `plockstat(1M)`, and `intrstat(1M)` where available (the last three are not yet available on all operating systems that have DTrace).

When a D program is executed (script or command line), the program is compiled into byte code,[9] representing the predicates and actions that can be bound to probes. The actual code is validated for safety and executed in the kernel in a vir-

9. This is similar to what happens when a program written in Java is compiled by the Java compiler.

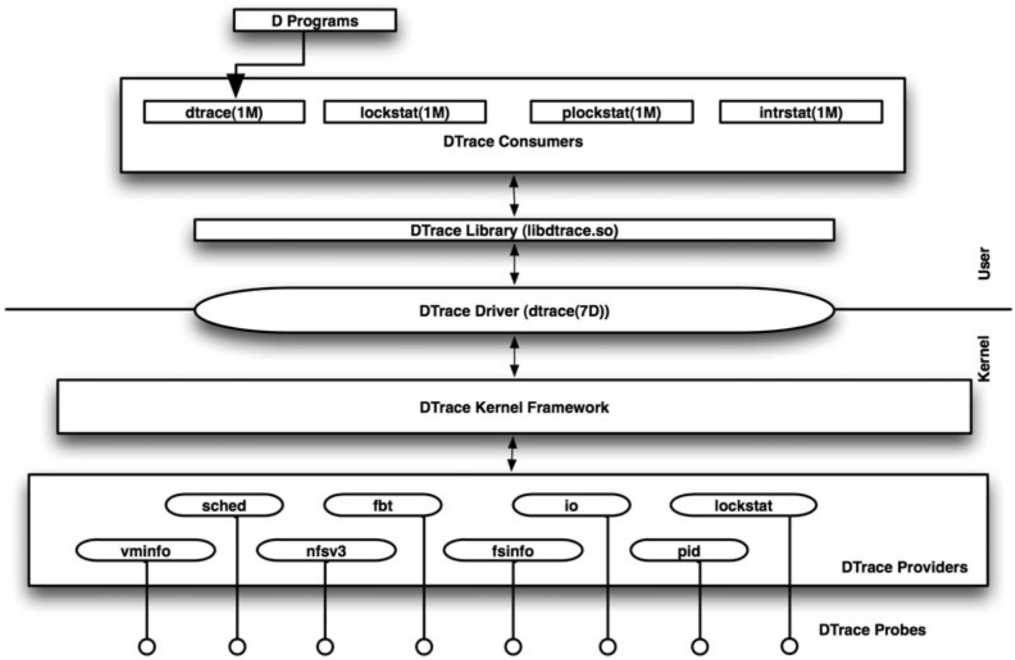


Figure 1-2 DTrace architecture

tual machine–like environment. That is, part of the kernel DTrace framework includes an emulated CPU supporting an RISC instruction set.

The internal implementation includes interfaces between the kernel framework and the providers. Since it is the providers that manage the probes, the framework calls into the providers based on the compiled D program to enable the requested probes. During the execution of the D program, requested data is collected, buffered, and returned to the requesting consumer. When the D program terminates, the providers disable the probes, restoring all the instrumented code paths to their original states.

Summary

DTrace is a revolutionary technology that provides observability up and down the entire software stack, without requiring code modifications, through the use of instrumentation points called probes. DTrace providers, a core component of the framework, manage probes and enhance observability by abstracting complex areas of the system with intuitively named probes and probe arguments that facilitate capturing relevant data. The D language and DTrace variables, functions, and subroutines combined provide a powerful and flexible environment for instrumenting and observing systems.

In this chapter, we introduced all aspects of DTrace: architecture, core components, and the D language. Examples demonstrated the use of DTrace probes and some of the available DTrace functions, subroutines, and variables. Throughout the remainder of this book, we will expand on all the material presented in this chapter, with an emphasis on DTrace by example.

2

D Language

The D programming language was inspired by C and awk(1), with built-in support for variables, strings, and a special data type called *aggregations*. This chapter summarizes the D programming language syntax in the abstract, as well as usage of the dtrace(1M) command; its use is extensively demonstrated in the numerous script examples throughout this book.

The Solaris Dynamic Tracing Guide[1] (often called the DTrace Guide) contains the complete reference for the D language and was released with Solaris 10 on the Sun Documentation Web site in HTML and PDF format, at more than 400 pages in length. It was later updated[2] and made available as an editable online wiki.[3] It covers all syntax, operators, and functions, as well as demonstrates each of the language components. As described in the preface, this book is intended as a complementary text to the DTrace Guide, by providing demonstrations of using DTrace to solve problems and a cookbook of complete scripts.

1. This is part number 817-6223, "Solaris Dynamic Tracing Guide," currently at *http://docs.sun.com/doc/817-6223*.
2. This is part number 819-3620, "Solaris Dynamic Tracing Guide," currently at *http://docs.sun.com/doc/819-3620*.
3. This is currently at *http://wikis.sun.com/display/DTrace/Documentation*.

This chapter will summarize the D language concisely, in a format inspired by an original paper for `awk`.[4] For the complete language reference, refer to the DTrace Guide.

D Language Components

In this section, we provide an overview of the main components of a D program.

Usage

The command

```
dtrace -n program
```

will execute the D program, instrumenting any probes described within it. The program can be saved to a file and executed using the following:

```
dtrace -s file.d
```

`file.d` is a D script, which for ease of identification has a `.d` extension. By placing an interpreter line at the top of the script

```
#!/usr/sbin/dtrace -s
```

the file can be made executable (`chmod a+x file.d`) and run like any other program:

```
./file.d
```

DTrace requires root (superuser) privileges to execute, unless finer-grained privileges are supported on the operating system and configured. For some systems,

4. "Awk: A Pattern Scanning and Processing Language (Second Edition)," Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Unix 7th Edition man pages, 1978. Online at *http://plan9.bell-labs.com/7thEdMan/index.html*.

the root shell can be used to launch DTrace directly, while for others the `sudo(8)` command may be preferable:

```
sudo ./file.d
```

Program Structure

A D program is a series of statements of the following form:

```
probes /predicate/ { actions }
probes /predicate/ { actions }
...
```

When probes fire, the predicate test determines whether to execute the actions (also called the *clause*), which are a series of statements. Without the predicate, the actions are always executed. Without a predicate or an action, a default line of output is printed to indicate that the probe fired. The only valid combinations are the following:

```
probes
probes { actions }
probes /predicate/ { actions }
```

The actions may be left blank, but when doing so, the braces are still necessary.

Probe Format

Probes are instrumentation points, described with this format:

```
provider:module:function:name
```

where

`provider` names the provider. Providers are libraries of related probes.

`module` describes the probe software location; this is either a kernel module or a user segment.

`function` names the software function that contains the probe.

`name` is the name of the probe from the provider.

The provider and name fields are terms to describe the probe, whereas the module and function fields explain the probe's software location. For some providers, it is these software locations that we want to instrument and are specified by the D programmer; for other providers, these fields are observational and are left unspecified (blank).

So, the probe description

```
syscall::read:entry
```

matches the `entry` probe from the `syscall` provider, when the function name is `read`. The syscall provider does not make use of the module field.

Predicates

Instead of conditional statements (`if/then/else`), DTrace has predicates. Predicates evaluate their expression and, if true, execute the action statements that follow in the clause. The expression is written in D, similar to the C language. For example, the predicate

```
/uid == 101/
```

will execute the action that follows only if the `uid` variable (current user ID) is equal to 101.

By not specifying a test

```
/pid/
```

the predicate will check that the contents are nonzero (`/pid/` is the same as `/pid != 0/`). These can be combined with Boolean operators, such as logical AND (`&&`), which requires both expressions evaluate true for the action to be taken:

```
/pid && uid == 101/
```

Actions

Actions can be a single statement or multiple statements separated by semicolons:

```
{ action one; action two; action three }
```

The final statement may also have a semicolon appended. The statements are written in D, similar to the C language, and can manipulate variables and call built-in functions. For example, the action

```
{ x++; printf("x is %d", x); }
```

increments a variable, `x`, and then prints it out.

Probes

Probes are made available by providers. Commonly available providers include dtrace,[5] for the `BEGIN` and `END` probes; profile, for `profile` and `tick` probes; and syscall, for system call `entry` and `return` probes. The full probe name is four fields separated by colons. See the other chapters of this book for more probes and providers, including Appendix C, Provider Argument Reference.

Wildcards

Wildcards (`*`) can be used in probe fields to match multiple probes. The field `foo*` matches all fields that begin with `foo`, and `*foo*` matches all that contain `foo`.

A field that is only a wildcard can be left blank to match everything. For example, to match a probe from any module or function, either of these will work:

```
provider:*:*:name
provider:::name
```

5. This is a provider that is called `dtrace`. (DTrace is the technology, and `dtrace(1M)` is the command.)

Blank fields to the left can be left out entirely; so these are equivalent:

```
:::name
::name
:name
name
```

To test wildcards, the `-l` option to `dtrace(1M)` can be used to list probes. This example

```
dtrace -ln 'syscall::read*:entry'
```

lists all `entry` probes from the syscall provider that have a function name beginning with `read` and for all module names. The probe name is put in single forward quotes to prevent the shell from attempting to interpret wildcards as shell metacharacters.

BEGIN and END

The dtrace provider has a `BEGIN` probe that fires at the start of the program and an `END` probe that fires at the end. The `BEGIN` probe can be used to initialize variables and print output headers, and the `END` probe can be used to print final reports.

profile and tick

The profile provider can create timed probes that fire at custom frequencies. The probe

```
profile:::profile-1234hz
```

fires on all CPUs at a rate of 1234 Hertz. The `profile-` probe may be used to sample what is executing systemwide. Apart from `hz`, other suffixes include `ms` (milliseconds), `s` (seconds), and `m` (minutes). The fastest sampling possible with

DTrace is 4999 Hertz. The profile provider also provides the `tick-` probe, for example,

```
profile:::tick-1s
```

which fires on one CPU only. The tick probe can be used for printing output summaries at the specified interval.

syscall Entry and Return

The syscall provider instruments the entry and return of system calls. The probe

```
syscall::read:entry
```

will fire when the `read(2)` system call begins, at which point the entry arguments to `read(2)` can be inspected. For example, use

```
syscall::read:entry { printf("%d bytes", arg2); }
```

to print the requested bytes, which are the third argument (`arg2`) to the `read(2)` system call.

And the probe

```
syscall::read:return
```

fires when the `read(2)` system call completes, at which point the return value and error status (`errno`) can be inspected. For example, to trace only errors showing the `errno` value, use this:

```
syscall::read:return /arg0 < 0/ { trace(errno); }
```

Variables

DTrace automatically instantiates variables on first assignment. The actions

```
a = 1;
b = "foo";
```

declare and assign an integer variable `a` and a string variable `b`. Without an explicit type cast, integer variables are of type `int` (signed 32-bit). Variables can be cast as in the C language:

```
a = (unsigned long long)1;
a = (uint64_t)1;
a = 1ULL;
```

These three examples are equivalent and declare the `a` variable to be an unsigned 64-bit integer, assigned a value of 1.

If variables are used before assignment, such as in predicates, their type is unknown, and an error will be generated. Either assign the variable beforehand, thus informing DTrace of the type, or cast the variables before use. Outside of any action group, the lines

```
int a;
string b;
```

will cast the variable `a` as an integer and `b` as a string.

Types

Integer variable types known by DTrace include the following:

- `char`: 8-bit character
- `short` or `int16_t`: Signed 16-bit integer
- `int` or `int32_t`: Signed 32-bit integer
- `long long` or `int64_t`: Signed 64-bit integer

`unsigned long long` or `uint64_t`: Unsigned 64-bit integer

Integer constants may use the suffixes `U` for unsigned and `L` for long.

Floating-point types (`float`, `double`, `long double`) may be used for tracing and formatting with `printf()`; however, operators cannot be applied.

String types are supported and use the same operators as other types. The predicate

```
/b == "foo"/
```

will return true (and execute the action clause) if the `b` string variable is equal to `foo`.

Strings are `NULL` terminated and, when empty, are equivalent to `NULL`. The example

```
/b != NULL/
```

tests that the string variable `b` contains data (not `NULL`).

Operators

All operators and order of precedence follow the ANSI-C conventions.

Arithmetic operators are supported for integers only. They are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulus). The expression

```
a = (b + c) * 2;
```

will add `b` and `c`, then multiply by 2, and finally assign the result to `a`.

Assignment and *unary* operators may be used as in C, such that these are equivalent:

```
x = x + 1;
x += 1;
x++;
```

Relational operators may be applied to integers, pointers, or strings; they are == (equal to), != (not equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). The predicate

```
/a > 2/
```

will fire the action clause if the a variable is greater than 2.

Boolean operators may also be used:

```
/a > 2 && (c == 3 || d == 4)/
```

The Boolean operators are && (AND), || (OR), and ^^ (XOR).

Bitwise operators are also supported: & (and), | (or), ^ (xor), << (shift left), and >> (shift right).

Ternary operators may be used for simple conditional expressions. The example

```
a = b >= 0 ? b : -b;
```

will assign the absolute value of b to a.

For the complete list of operators, see Appendix B, D Language Reference.

Scalar

Scalar variables store individual values. They are known globally and can be accessed from any action clause. The assignment

```
a = 1;
```

assigns the value 1 to the scalar variable a.

Scalars can be accessed by probes firing on multiple CPUs simultaneously and as such may become corrupted. Their use is therefore discouraged whenever other variable types (thread-local variables or aggregations) can be used instead.

Associative Arrays

Associative arrays can contain multiple values accessed via a key. They are declared with an assignment of the following form:

```
name[key] = expression;
```

The key may be a comma-separated list of expressions. The example

```
a[123, "foo"] = 456;
```

declares a key of the integer 123 and the string `foo`, storing the integer value 456.

Associative arrays have the same issues as scalars, with the potential to become corrupted if multiple CPUs modify the same key/value simultaneously.

Structs and Pointers

The D language supports structures and pointer operations based on C (ANSI-C). Structures can be defined in `typedef struct` statements outside of action clauses, and header files can be included (`#include <file.h>`) when the C preprocessor (`-C` option) is used. Many structures are already known by DTrace. The example

```
curpsinfo->pr_psargs
```

is a built-in type of `struct psinfo`, defined under `/usr/lib/dtrace`. And this example

```
fbt::fop_create:entry { trace(stringof(args[0]->v_path)); }
```

is possible only when the `fbt` provider has access to kernel type data so that the `args[]` array can be aware of the types, including structures, of the probe arguments. This example was from Oracle Solaris, where the fbt provider knows that the first argument to `fop_create()` is a `vnode_t`, allowing the `v_path` member to be retrieved.

Thread Local

Thread-local variables are stored with the current thread of execution. They have this prefix:

```
self->
```

To declare a thread-local variable, set a value. The example

```
self->x = 1;
```

declares a thread-local variable, x, to contain the value 1.

To free a thread-local variable, set it to zero (also for string variables):

```
self->x = 0;
```

If thread-local variables are set but not freed after use, memory may be consumed needlessly while those threads still exist on the system. Once the thread is destroyed, the thread-local variables are freed.

Thread-local variables should be used in preference to scalars and associative arrays wherever possible to avoid the possibility of multiple CPUs writing to the same D variable location and the contents becoming corrupted. They may also improve performance, since thread-local variables can be accessed by only one CPU (one thread) at a time.

Clause Local

Clause-local variables are for use within a single of action group { }. They have this prefix:

```
this->
```

To declare a clause-local variable, set a value. The example

```
this->y = 1;
```

declares a clause-local variable, y, to contain the value 1.

Clause-local variables do not need to be freed; this is done automatically when the probe finishes executing all actions associated with that probe. If there are multiple `probe { action }` groups for the same probe, clause-local variables can be accessed across the action groups.

They should be used for temporary variables within an action, because they have the lowest performance cost of all the variable types.

Built-in

A variety of built-in variables are available as scalar globals. They include the variables presented in Table 2-1.

Table 2-1 Built-in Variables

Variable Name	Type	Description
`arg0...arg9`	uint64_t	Probe arguments; content is provider-specific
`args[]`	*	Typed probe arguments; content is provider-specific
`cpu`	processorid_t	CPU ID of the current CPU
`curpsinfo`	psinfo_t	Process state info for the current thread
`curthread`	kthread_t	Operating system internal kernel thread structure for the current thread
`errno`	int	Error value from the last system call
`execname`	string	The name of the current process
`pid`	pid_t	Process ID for current process
`ppid`	pid_t	Parent process ID for current process
`probeprov`	string	Provider name of the current probe
`probemod`	string	Module name of the current probe
`probefunc`	string	Function name of the current probe
`probename`	string	Name of the current probe
`stackdepth`	uint_t	Current thread's stack frame depth
`tid`	id_t	Thread ID of the current thread
`timestamp`	uint64_t	Elapsed time since boot in nanoseconds
`uid`	uid_t	Real user ID of the current process
`uregs[]`	uint64_t	The current thread's saved user-mode register values
`vtimestamp`	uint64_t	Current thread's on-CPU time in nanoseconds
`walltimestamp`	uint64_t	Nanoseconds since epoch (January 1, 1970)

It is common to use built-in variables in predicates in order to gather data for specific processes, threads, or events of interest. The example

```
/execname == "ls"/
```

is a predicate that will cause the actions in the clause to be executed only when the process name is `ls`.

Macro

DTrace provides macro variables including the ones presented in Table 2-2.

For example, a D script called `file.d` could match the `$target` process ID in a predicate:

```
/pid == $target/
```

which is provided to the D script at the command line,

```
# ./file.d -p 123
```

so the predicate will fire the action clause only if the specified process ID, 123, is on-CPU.

Table 2-2 Macro Variables

Variable Name	Type	Description
`$target`	`pid_t`	Process ID specified using `-p PID` or `-c command`
`$1..$N`	Integer or string	Command-line arguments to `dtrace(1M)`
`$$1..$$N`	String (forced)	Command-line arguments to `dtrace(1M)`

External

External variables are defined by the operating system (external to DTrace) and accessed by prefixing the kernel variable name with a backquote. The kernel integer variable `k` could be printed using this:

```
printf("k: %d\n",`k);
```

Aggregations

Aggregations are a special variable type used to summarize data. They are prefixed with an at (`@`) sign and are populated by *aggregating functions*. The action

```
@a = count();
```

populates an aggregation, `a`, that counts the number of times it was invoked. Although this sounds similar to a scalar `a` with the operation `a++`, global scalars may suffer data corruption from simultaneous writing across CPUs, as well as a performance penalty for accessing the same location. Aggregations avoid this by populating data in per-CPU buffers, which are combined when printing.

Aggregations can be printed and emptied explicitly with the `printa()` and `trunc()` functions (covered later in the chapter). Aggregations not explicitly printed or truncated are automatically printed at the end of D programs. They cannot be tested in predicates.

Aggregations may have keys, like associative arrays. The example

```
@a[pid] = count();
```

will count events separately by `pid` (process ID). The aggregation will be printed as a table with the keys on the left and values on the right, sorted on the values in ascending order.

An aggregation without a name, `@`, may be used for D programs (especially one-liners) that use only one aggregation and so don't need a name to differentiate them.

Table 2-3 Aggregating Functions

Function	Arguments	Result
`count`	None	The number of times called.
`sum`	Scalar	The total value.
`avg`	Scalar	The arithmetic average.
`min`	Scalar	The smallest value.
`max`	Scalar	The largest value.
`stddev`	Scalar	The standard deviation.
`lquantize`	Scalar, lower bound, upper bound, step	A linear frequency distribution, sized by the specified range, of the values of the specified expressions. Increments the value in the *highest* bucket that is *less* than the specified expression.
`quantize`	Scalar	A power-of-two frequency distribution of the values of the specified expressions. Increments the value in the *highest* power-of-two bucket that is *less* than the specified expression.

Types

Table 2-3 lists functions that populate aggregations. There are four additional functions to manipulate aggregations: `trunc()`, `clear()`, `normalize()`, and `printa()`.

The example

```
@t = sum(x);
```

populates the aggregation `t` with the sum of the `x` value.

quantize()

The `quantize()` function populates a power-of-two frequency distribution. The action

```
@a = quantize(x);
```

populates the quantize aggregation, `a`, with the value `x`. This is printed as a distribution plot showing power-of-two ranges on the left, counts on the right, and a text rendition of the distribution. Because this is a data visualization, it is best explained with an example output:

```
  value  ------------- Distribution ------------- count
      0 |                                         0
      1 |@@@@                                     62
      2 |@                                        10
      4 |                                         5
      8 |                                         6
     16 |                                         3
     32 |@@@                                      46
     64 |@@@@@@@@@@@@@@@@@@@@@@                   316
    128 |@@@@@@@@                                 113
    256 |@                                        9
    512 |                                         1
   1024 |                                         0
```

The `value` column shows the minimum of the range, and the `count` shows the number in that range. The most common range while tracing in this example was 64–127, with a count of 316.

lquantize()

The `lquantize()` function populates a linear frequency distribution. The action

```
@a = lquantize(x, 0, 100, 10);
```

populates the linear quantize aggregation, `a`, with the value `x`. The other arguments set a minimum value of 0, a maximum of 100, and a range step of 10. Example output from this

```
  value  ------------- Distribution ------------- count
    < 0 |                                         0
      0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      27
     10 |                                         0
     20 |@                                        1
     30 |                                         0
     40 |                                         0
     50 |                                         0
     60 |@@@@                                     3
     70 |                                         0
```

shows the size of each range is 10 in the `value` column. This example shows the most frequent range while tracing was 0–9, with a count of 27.

trunc() and clear()

The `trunc()` function can either completely clear an aggregation, leaving no keys

```
trunc(@a);
```

or truncate an aggregation to the top number of keys specified. For example, this function truncates to the top 10:

```
trunc(@a, 10);
```

The `clear()` function clears the values of keys but leaves the keys in the aggregation.

normalize()

The `normalize()` function can divide an aggregation by a value. The example

```
normalize(@a, 1024);
```

will divide the `@a` aggregation values by 1,024; this may be used before printing to convert values to kilobytes instead of bytes.

printa()

The `printa()` function prints an aggregation during a D program execution and is similar to `printf()`. For example, the aggregation

```
@a[x, y] = sum(z);
```

where the key consists of the integer `x` and string `y`, may be printed using

```
printa("%10d %-32s %@8d\n", @a);
```

which formats the key into columns: `x`, 10 characters wide and right-justified; `y`, 32 characters wide and left-justified. The aggregation value for each key is printed using the `%@` format code, eight characters wide and right-justified.

The `printa()` function can also print multiple aggregations:

```
printa("%10s %@8d %@8d\n", @a, @b);
```

The aggregations must share the same key to be printed in the same `printa()`. By default, sorting is in ascending order by the first aggregation. Several options exist for changing the default sort behavior and for picking which aggregation to sort by.

`aggsortkey`: Sort by key order; any ties are broken by value.

`aggsortrev`: Reverse sort.

`aggsortpos`: Position of the aggregation to use as primary sort key.

`aggsortkeypos`: Position of key to use as primary sort key.

The aggregation sort options can be used in combination. See the "Options" section for setting options.

Actions

DTrace actions may include built-in functions to print and process data and to modify the execution of the program or the system (in a carefully controlled manner). Several key functions are listed here.

Actions that print output (for example, `trace()` and `printf()`) will also print default output columns from DTrace (CPU ID, probe ID, probe name), which can be suppressed with quiet mode (see "Options" section). The output may also become shuffled on multi-CPU systems because of the way DTrace collects per-CPU buffers and prints them out, and a time stamp field can be included in the output for postsorting, if accurate; chronological order is required.

trace()

The `trace()` action takes a single argument and prints it:

```
trace(x)
```

This prints the variable `x`, which may be an integer, string, or pointer to binary data. DTrace chooses an appropriate method for printing, which may include printing hexadecimal (hex dump).

printf()

Variables can be printed with formatting using `printf()`, based on the C version:

```
printf(format, arguments ...)
```

The format string can contain regular text, plus directives, to describe how to format the remaining arguments. Directives comprise the following.

`%`: To indicate a format directive.

`-`: (Optional.) To change justification from right to left.

`width`: (Optional.) Width of column as an integer. Text will overflow if needed.

`.length`: (Optional.) To truncate to the length given.

`type`: Covered in a moment.

Types include the following.

`a`: Convert pointer argument to kernel symbol name.

`A`: Convert pointer argument to user-land symbol name.

`d`: Integer (any size).

`c`: Character.

`f`: Float.

`s`: String.

`S`: Escaped string (binary character safe).

`u`: Unsigned integer (any size).

`Y`: Convert nanoseconds since epoch (walltimestamp) to time string.

For example, the action

```
printf("%-8d %32.32s %d bytes\n", a, b, c);
```

prints the `a` variable as an integer in an 8-character-wide, left-justified column; the `b` variable as a string in a 32-character-wide, right-justified column, and with no overflow; and the `c` variable as an integer, followed by the text `bytes` and the new line character `\n`.

For the complete `printf()` reference, see Appendix B, D Language Reference.

tracemem()

To print a region of memory, the `tracemem()` function can be used. The example

```
tracemem(p, 256);
```

prints 256 bytes starting at the `p` pointer, in hexadecimal. If `tracemem()` is given a data type it can recognize, such as a `NULL`-terminated string, it will print that in a meaningful way (not as a hex dump).

copyin()

DTrace operates in the kernel address space. To access data from the user-land address space associated with a process, `copyin()` can be used. The example

```
a = copyin(p, 256);
```

copies 256 bytes of data from the `p` user-land pointer into the variable `a`. The buffer pointers on the `read(2)` and `write(2)` syscalls are examples of user-land pointers, so that

```
syscall::write:entry { w = copyin(arg0, arg2); }
```

will copy the data from `write(2)` into the `w` variable.

stringof() and copyinstr()

To inform DTrace that a pointer is a string, use `stringof()`. The example

```
printf("%s", stringof(p));
```

treats the `p` pointer variable as a string and prints it out using `printf()`.

`stringof()` works only on pointers in the kernel address space; for user-land pointers, use `copyinstr()`. For example, the first argument to the `open(2)` syscall is a user-land pointer to the path; it can be printed using the following:

```
syscall::open:entry { trace(copyinstr(arg0)); }
```

This may error if the pointer has not yet been faulted into memory; if this becomes a problem, perform the `copyinstr()` after it has been used (for example, on `syscall::open:return`).

strlen() and strjoin()

Some string functions are available for use in D programs, including `strlen()` and `strjoin()`. The example

```
strjoin("abc", "def")
```

returns the string `abcdef`. Apart from literal strings, this may also be used on string variables.

stack(), ustack(), and jstack()

The `stack()` action fetches the current kernel stack back trace. Used alone,

```
stack();
```

prints out the stack trace when the probe fires, with a line of output per stack frame. To print a maximum of five stack frames only, use this:

```
stack(5);
```

It can also be used as keys for aggregations. For example, the action

```
@a[stack()] = count();
```

counts invocations by stack trace; that is, when the aggregation is printed, a list of stack traces will be shown along with the counts for each stack, in ascending order. To print them in `printa()` statements, use the `%k` format directive.

The `ustack()` action fetches the current user-stack backtrace. This is stored as the addresses of functions, which are translated into symbols when printed out; however, if the process has terminated by that point, only hexadecimal addresses will be printed.

The `jstack()` action behaves similarly to `ustack()`; however, it may insert native-language stack frames when available, such as Java classes and methods from the JVM.

Refer to the "Options" section in this chapter for tunable options that apply to the stack functions.

sizeof()

The `sizeof()` operator returns the size of the data type, in bytes. The example

```
sizeof (uint64_t)
```

returns the number 8 (bytes).

exit()

This exits the D program with the specified return value. To exit and return success, use the following:

```
exit(0);
```

Speculations

Speculative tracing provides the ability to tentatively trace data and then later decide whether to *commit* it (print it out) or *discard* it. The action

```
self->maybe = speculation();
```

creates a speculative buffer and saves its identifier in `self->maybe`. Then, the action clause

```
{ speculate(self->maybe); printf("%d %d", a, b); }
```

prints the integer variables `a` and `b` into that speculative buffer. Finally,

```
/errno != 0/ { commit(self->maybe); }
```

will commit the speculation, printing all the contents of that speculative buffer. In this case, when the `errno` built-in is nonzero (probe description not shown), the example

```
/errno == 0/ { discard(self->maybe); }
```

will discard the contents of the speculation.

Translators

Translators are special functions that convert from one data type to another. They are used by some stable providers to convert from unstable kernel locations into the stable argument interface, as specified in the `/usr/lib/dtrace` files. The example

```
xlate <info_t *>(a)->name;
```

takes the `a` variable and applies the appropriate `info_t` translator to retrieve the `name` member; the translator is chosen based on the type of `a`. For example, if `a` were of type `_impl_t`, the following translator would be used (defined earlier),

```
translator info_t < _impl_t *I > {
            name = stringof(I->nm);
            length = I->len;

}
```

which translates type `_impl_t` (not defined here) into the members defined earlier.

Custom translators can be written and placed in additional `.d` files in the `/usr/lib/dtrace` directory, which will be automatically loaded by `dtrace(1M)` for use in D programs.

Others

For the complete list of actions, see Appendix B, D Language Reference. These include `basename()`, `bcopy()`, `dirname()`, `lltostr()`, `progenyof()`, and `rand()`. These also include the destructive (`-w` required) actions: `stop()`, `raise()`, `copyout()`, `copyoutstr()`, `system()`, and `panic()`.

Options

There are various options to control the behavior of DTrace, which can be listed using this:

```
# dtrace -h
```

The options include the following:

- `-c command`: Runs command and exit on completion, setting `$target` to its PID
- `-n probe`: Specifies a probe
- `-o file`: Appends output to the file
- `-p PID`: Provides PID as `$target` and exits on completion
- `-q`: Suppresses default output
- `-s file`: Executes the script file
- `-w`: Allows destructive actions
- `-x option`: Sets `option`

Some of these can be specified in the D script as `pragma` actions. The example

```
#pragma D option quiet
```

will set quiet mode (`-q`). And the example

```
#pragma D option switchrate=10hz
```

sets the buffer switch rate to 10 Hertz (which can decrease the latency for traced output). These name=value options can also be set at the command line using `-x`. Others include the following:

- `bufsize`: Principal buffer size
- `defaultargs`: If unspecified, `$1` becomes 0 and `$$1` becomes “”
- `destructive`: Allow destructive actions
- `dynvarsize`: Dynamic variable space size
- `flowindent`: Indent output on function entry

`strsize`: Max size of strings

`stackframes`: Max number of stack frames in `stack()`

`ustackframes`: Max number of user stack frames in `ustack()`

`jstackstrsize`: Size of the string buffer

For the complete list of tunables, see Appendix A, DTrace Tunables.

Example Programs

Here are a few example one-liners and a script, chosen to demonstrate components of the D language. The other chapters in this book have many more examples to consider.

Hello World

This example uses the `dtrace` provider `BEGIN` probe to print a text string at the start of execution. Ctrl-C was hit to exit `dtrace(1M)`:

```
# dtrace -n 'BEGIN { trace("Hello World!"); }'
dtrace: description 'BEGIN ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN   Hello World!
^C
```

Apart from our text, DTrace has printed a line to describe how many probes were matched, a heading line, and then the CPU ID, probe ID, and function:name component of the probe. This default output can be suppressed with quiet mode (`-q`).

Tracing Who Opened What

This traces the `open(2)` syscall, printing the process name and path:

```
# dtrace -n 'syscall::open:entry { printf("%s %s", execname, copyinstr(arg0)); }'
dtrace: description 'syscall::open:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  96337                       open:entry nscd /etc/inet/ipnodes
  1  96337                       open:entry nscd /etc/resolv.conf
  1  96337                       open:entry nscd /etc/hosts
  1  96337                       open:entry nscd /etc/resolv.conf
  1  96337                       open:entry automountd /var/run/syslog_door
  1  96337                       open:entry automountd /dev/udp
  1  96337                       open:entry automountd /dev/tcp
```

```
  1  96337                       open:entry sh /var/ld/ld.config
  1  96337                       open:entry sh /lib/libc.so.1
^C
```

Tracing fork() and exec()

Fundamental to process creation, these system calls can be traced along with the `pid` and `execname` built-ins to see their behaviors:

```
# dtrace -n 'syscall::fork*: { trace(pid); }'
dtrace: description 'syscall::fork*: ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  13072                    forksys:entry     16074
  0  13073                   forksys:return     16136
  0  13073                   forksys:return     16074
^C

# dtrace -n 'syscall::exec*: { trace(execname); }'
dtrace: description 'syscall::exec*: ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  12926                      exece:entry   bash
  0  12927                     exece:return   ls
^C
```

Counting System Calls by a Named Process

An aggregation is used to count system calls from Mozilla Firefox, which is running with the process name `firefox-bin`:

```
# dtrace -n 'syscall:::entry /execname == "firefox-bin"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 237 probes
^C

  close                                                              1
  setsockopt                                                         1
  getpid                                                             2
  yield                                                             47
  writev                                                           130
  lwp_park                                                         395
  ioctl                                                            619
  write                                                           1102
  pollsys                                                         1176
  read                                                            1773
```

Showing Read Byte Distributions by Process

Distribution plots can summarize information while retaining important details; this shows power-of-two distributions for `read(2)` return values:

```
# dtrace -n 'syscall::read:return { @[execname] = quantize(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
^C
[...output truncated...]

  firefox-bin
           value  ------------- Distribution ------------- count
              -2 |                                         0
              -1 |@@@@                                     25
               0 |                                         0
               1 |@@@@@@@@@@                               64
               2 |                                         0
               4 |                                         2
               8 |                                         2
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@                166
              64 |                                         2
             128 |                                         0
             256 |                                         2
             512 |                                         0

  Xorg
           value  ------------- Distribution ------------- count
              -2 |                                         0
              -1 |@@@@@@@@@@@@@@@                          233
               0 |                                         0
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |@@@@@@@                                  100
              16 |@@@@@@@                                  106
              32 |@@@@                                     59
              64 |@                                        12
             128 |@                                        8
             256 |@                                        10
             512 |@                                        13
            1024 |                                         4
            2048 |@@@                                      42
            4096 |@                                        22
            8192 |                                         0
```

This shows a trimodal distribution for Firefox, with 25 returns of -1 (error), 64 of 1 byte, and 166 of between 32 and 63 bytes.

Profiling Process Names

The `profile` probe (from the profile provider) can be used to sample across all CPUs at a specified rate. Here the process name is sampled and recorded in a `count()` aggregation at 997 Hertz, on a 2x CPU system. A tick probe is used to print the aggregation and then truncate all data every second so that interval prints only the data from the last second.

```
# dtrace -n 'profile-997 { @[execname] = count(); } tick-1s { printa(@); trunc(@); }'
dtrace: description 'profile-997 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  21301                         :tick-1s
```

```
  Shades                                                      1
  Preview                                                     2
  VBoxNetDHCP                                                 3
  ntpd                                                        4
  dtrace                                                      5
  Adium                                                       6
  VBoxSVC                                                     6
  quicklookd                                                  8
  soffice                                                    11
  VirtualBox                                                 13
  Terminal                                                   14
  SystemUIServer                                             19
  WindowServer                                               35
  firefox-bin                                               856
  kernel_task                                              1008

  1  21301                       :tick-1s
  Preview                                                     1
  dtrace                                                      1
  ntpd                                                        1
  VBoxSVC                                                     3
  Adium                                                       4
  soffice                                                    14
  VirtualBox                                                 15
  SystemUIServer                                             18
  WindowServer                                               19
  Terminal                                                   30
  firefox-bin                                               883
  kernel_task                                              1005
[...]
```

Timing a System Call

Most function calls will return from the same thread that they enter,[6] so a thread-local variable can be used to associate these events. Here a time stamp is saved on the `write(2)` entry so that the time can be calculated on return:

```
# dtrace -n 'syscall::write:entry { self->s = timestamp; }
    syscall::write:return /self->s/
    { @["ns"] = quantize(timestamp - self->s); self->s = 0; }'
dtrace: description 'syscall::write:entry ' matched 2 probes
^C

  ns
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@                                        6
            8192 |@@@@@                                    22
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            122
           32768 |@@@                                      13
           65536 |@                                        4
          131072 |                                         2
          262144 |                                         0
```

6. Exceptions include `fork()`, which has one entry and two returns, as shown in 7.3.

The output showed that most writes took between 8 and 32 microseconds. The thread-local variable was just `self->s`, which is sufficient for timing system calls (but not for recursive functions, because multiple entry probes will overwrite `self->s`).

Snoop Process Execution

This script prints details on new processes as they execute:

```
1       #!/usr/sbin/dtrace -s
2
3       #pragma D option quiet
4       #pragma D option switchrate=10hz
5
6       dtrace:::BEGIN
7       {
8             printf("%-20s %6s %6s %6s  %s\n", "ENDTIME",
9                 "UID", "PPID", "PID", "PROCESS");
10      }
11
12      proc:::exec-success
13      {
14            printf("%-20Y %6d %6d %6d  %s\n", walltimestamp,
15                uid, ppid, pid, execname);
16      }
```

Script pexec.d

Line 1 is the interpreter line, feeding the following script `-s` to `/usr/sbin/dtrace`.

Line 3 instructs DTrace to *not* print the default output when probes fire.

Line 4 increases the switch rate tunable to 10 Hertz so that output is printed more rapidly.

Lines 8 and 9 print a heading line for the output, fired during the `dtrace:::BEGIN` probe.

Line 12 uses the `proc:::exec-success` probe to trace successful execution of new processes.

Lines 14 and 15 print details of the new process from various built-in variables, formatted in columns using `printf()`.

Running the script yields the following:

```
# ./pexec.d
ENDTIME                  UID   PPID    PID  PROCESS
2010 Jul 20 22:16:04     501    244  38131  ps
2010 Jul 20 22:16:04     501    244  38132  grep
2010 Jul 20 22:16:38     501    159  38129  nmblookup
2010 Jul 20 22:16:39     501    159  38130  nmblookup
^C
```

For this example, the command `ps -ef | grep firefox` was executed in another terminal window, which was traced by `pexec.d`. The script was left tracing, and it caught the execution of an unexpected process, `nmblookup`, which ran twice.

Summary

In this chapter, we summarized the D language, including the syntax, operators, and built-in functions; you can find full reference tables for these in Appendixes A and B. The remaining chapters continue to demonstrate D, as used in one-liners and scripts.

3

System View

Now that we've introduced you to DTrace and covered the D language, it's time to get into what DTrace is really all about—solving problems and understanding workload behavior. Some problems can be quickly correlated to a specific area of hardware or software, but others have a potentially broader source of root causes. These require starting with a *systemwide* view and drilling down based on what the first-pass analysis reveals. In this chapter, we'll show how existing non–DTrace-based tools can help make that first pass and how DTrace can then complement them and take analysis further. Specific hardware subsystems, networking, disk I/O, file system, and specific applications are covered in greater detail in later chapters.

So, what do we mean by *system* and *system view*? We're referring to a coarse decomposition of the various components that make up your computer system. Specifically, looking at the major hardware subsystems, we're talking about the following:

- Processors (CPUs), including modern multicore processors
- Memory, in other words, installed physical RAM
- Disk I/O, including controllers and storage
- Network I/O

From a software point of view, there's the operating system (kernel) and userland application software. We'll show techniques for analyzing user software here, as well as figuring out what the kernel is doing. For some applications, we'll explore specific types of software that have a significant installed base and use in dedicated chapters.

Start at the Beginning

One of the challenges facing new DTrace users is where to begin. DTrace is an extremely powerful and potentially complex tool; it's up to you to decide what to do with it. Given that DTrace can examine the entire operating system plus applications, simply picking a starting point for analysis can be daunting. This is especially true when DTrace is used to troubleshoot application and performance problems in production environments, where time is of the essence.

The basic approach to any problem starts with the same first step—defining the problem in terms of something that can be measured. Examples in the domain of performance include transaction response time and time to run batch jobs or other workload tasks. The commonality here is *time*, a primary metric for quantifying performance issues. Other metrics such as high CPU utilization, low network bandwidth, high disk utilization, and so on, are not performance problems per se; they may be contributing to the actual problem, but first and foremost the problem needs to be defined in the context of what the workload is requesting and how long it is taking, not in terms of the utilization of the components that service it.

Even with a solid problem definition, determining the underlying cause (or causes—there may be several contributing factors) sometimes requires taking a big-picture view of the system and drilling down based on observations and analysis. A system view starts with bundled tools and utilities that provide data on running processes and hardware utilization. That is, DTrace may not be the first tool you should use. You could start with DTrace to get much of the system view data, but it can be easier to start with your favorite set of "stat" tools to get the big picture and drill down from there.

To continue getting started, the next sections summarize performance analysis in the abstract and then existing tool sets for the different operating systems. This leverages existing methodology and tools and allows us to focus on using DTrace to take analysis further. Another resource for Solaris performance analysis is *Solaris Performance and Tools* (McDougall, Mauro, and Gregg, 2006), which covers the Solaris tool set in detail.

System Methodology

You can use the following questions as guidelines to approaching an undefined performance issue and to help you on your way to finding the root cause. Most of these questions are operating system–generic, but some are Solaris-specific.

How busy are the CPUs?

- Is there idle time (%idle)?
- Are CPU cycles being consumed in user mode (%usr) or in the kernel (%sys)?
- Is CPU utilization relatively flat, or does it fluctuate?
- Is CPU utilization balanced evenly across the available CPUs?
- What code paths are making the CPUs busy?
- Are interrupts consuming CPU cycles?
- Are the CPU cycles spent executing code or stalled on bus (memory) I/O?
- Is the dispatcher run queue consistently nonzero?
- Does the dispatcher run queue depth fluctuate, with bursts of runnable threads?

Does the system have sufficient memory?

- Is the system "swapping" (or "paging") due to memory pressure?
- Is the page scanner running?
- How much memory are processes consuming (RSS)?
- Does the system make use of shared memory?
- How much time are applications spending allocating and freeing memory?
- How much time are applications waiting for memory pages to be paged in?
- Is there a file system component to the workload, where the file system page cache needs to be factored in as a potentially significant memory consumer?

How much disk I/O is the system generating?

- What is the average disk I/O service time?
- Does any disk I/O return with very high latency?
- Is the disk access pattern random or sequential?
- Are the I/Os generally small (less than 8KB) or large?
- Is there an imbalance in the disk I/O load, with some disks getting a much larger percentage of reads and writes than others?
- Are there disk I/O errors?

- Can a disk I/O load generation tool be used to investigate disk I/O performance and latency (outside the context of the applications)?

How much network I/O is occurring?

- Is the network load at or near the theoretical limit for the NIC, in terms of either bandwidth or packet rates?
- With which remote hosts is the system establishing TCP connections?
- Which remote hosts are causing the most network I/O?
- Are there multiple NICs?
- Is multipathing configured?
- How much time are the applications spending waiting for network I/O?
- Can network load generation tools be used to assess network throughput and latency (outside the context of the applications)?

The sections that follow in this chapter begin answering these questions for the CPU, memory, and disk and network resources, first using existing tool sets and then leading into DTrace. These questions will continue to be answered throughout the book in chapters dedicated to these topics (for example, Chapter 4, Disk I/O) or for consumers of these resources (for example, Chapter 12, Kernel, for CPU utilization by the kernel).

In some cases, you know the resource or subsystem to examine based on the problem or metrics previously examined. When it comes to available system resources, the high-level questions you must answer are the same.

How utilized is the resource?

Is the resource saturated with work or under contention?

Is the resource encountering errors?

What is the response time for the resource?

What workload abstractions (files/clients/requests) are consuming the resource?

Which system components (processes/threads) are consuming the resource?

Is the performance of the resource a result of workload applied (high load) or system implementation (poor configuration)?

System Tools

The tools and utilities available for examining system load and utilization metrics will vary across different operating systems. In some cases, the same utilities may be available on multiple platforms (for example, `sar(1M)`), or different systems may have utilities with similar names but generate very different output (for example, `vmstat(1M)` on Solaris vs. `vm_stat(1)` on Mac OS X). Always reference the appropriate man pages to determine available options and definitions of the output generated. Table 3-1 presents the most commonly used system tools.

Table 3-1 System Tools

Solaris	
Utility	**Description**
`sar(1)`	General-purpose System Activity Reporter providing numerous system statistics
`vmstat(1M)`	Reports virtual memory statistics and aggregates systemwide CPU utilization
`mpstat(1M)`	Per-CPU statistics
`iostat(1M)`	Disk I/O statistics
`netstat(1M)`	Network statistics
`kstat(1M)`	All available kernel statistics
`prstat(1M)`	Process/thread statistics
Mac OS X	
Utility	**Description**
`sar(1)`	General-purpose System Activity Reporter providing numerous system statistics
`vm_stat(1)`	Virtual memory statistics
`top(1)`	Process statistics
FreeBSD	
Utility	**Description**
`systat(1)`	Various system statistics
`vmstat(8)`	Virtual memory statistics
`iostat(8)`	Disk I/O statistics
`netstat(8)`	Network statistics
`sockstat(1)`	Open socket information
`procstat(1)`	Detailed process information
`top(1)`	Process statistics

The information in Table 3-1 is not a comprehensive list of every available utility for each operating system but are those most commonly used for a high-level system view. Solaris, for example, includes a large number of process-centric tools not listed here. Mac OS X includes a GUI-based Activity Monitor and other utilities useful for monitoring a system.

Observing CPUs

CPU utilization as a key capacity metric has years of history in IT and is reported by many of the traditional "stat" tools (Solaris `vmstat(1M)`, and so on). The actual meaning and usefulness of CPU utilization as a metric has diminished with the evolution of processor technology. Multiprocessor systems, processors with multiple execution cores, processor cores with multiple threads (or strands), processor cores with multiple execution units (integer, floating point) allowing multiple threads to advance instructions concurrently, and so on, all skew the notion of what CPU utilization level really means in any given operating system.

A specific example of this is memory bus I/O. CPU "load" and "store" instructions may stall while on-CPU, waiting for the memory bus to complete a data transfer. Since these stall cycles occur during a CPU instruction, they're treated as utilized, although perhaps not in the expected way (utilized *while* waiting!).

The level of parallelism of the workload is also a factor; a single-threaded application may consume 100 percent of a single CPU, leaving other CPUs virtually idle. In such a scenario, on systems with a large number of CPUs, tools that aggregate utilization would indicate very low systemwide CPU utilization, potentially steering you away from looking more closely at CPUs. A highly threaded workload with lock contention in the application may show many CPUs running at 100 percent utilization, but most of the threads are spinning on locks, rather than doing the work they were designed to do. The key point is that CPU utilization alone is not sufficient to determine to what extent the CPUs themselves are the real performance problem.

It is instructive to know exactly what the CPUs are doing—whether that is the processing of instructions or memory I/O stall cycles—and for what applications or kernel software.

CPU Strategy

Conventional tools and utilities can be a good place to start for a quick look at the CPUs. `vmstat(1)` provides one row of output per sample, so for multiprocessor systems, CPU utilization is aggregated into one set of usr/sys/idle metrics. This is especially useful for a high-level view on systems with a large number of CPUs; modern high-end systems can have hundreds of CPUs, making per-CPU utilization analysis daunting to start with. The `mpstat(1)` utility provides a row of output for each CPU, along with many other useful statistics, such as per-second counts of interrupts, system calls, context switches, and so on. The primary metric of interest when doing a first-pass system view is where CPU cycles are being consumed: how much of the busy time is in the kernel (%sys) vs. executing in user mode (%usr).

CPU Checklist

The checklist in Table 3-2 describes the high-level issues around CPU usage and performance.

Table 3-2 CPU Checklist

Issue	Description
Utilization—high %sys	The system is spending what appears to be an inordinate amount of time in the kernel. This may or may not be a problem—some workloads are kernel intensive (for example, NFS/file servers, network-intensive workloads, and so on). A kernel profile is the first step to determine where in the kernel CPU cycles are being consumed.
Utilization—high %user	When burning CPU cycles, for most applications, it's generally good to be spending most of the time in user mode. But that leaves the question of whether the cycles are being spent getting real work done or potentially spinning on user locks or some other area of code that's not advancing the workload. A profile of where the threads are spending time can answer this question.
Wait time	Are threads waiting for an available CPU? This is known as *run queue latency* and is easily tracked in Solaris with `prstat -Lm`, monitoring the `LAT` column. DTrace can be used to measure how much time threads are spending on run queues, waiting to run. The `vmstat(1M) r` column shows systemwide runnable threads.
Configuration	Solaris provides resource management tools such as processor sets, resource pools, CPU management, and different scheduling classes and priority control mechanisms that can affect CPU usage.

continues

Table 3-2 CPU Checklist (*Continued*)

Issue	Description
Interrupt load	Modern I/O devices, especially 10Gb network cards, can generate a high level of interrupts to the CPUs. If application threads are sharing those CPUs, they may be getting pinned frequently by the interrupt threads, throttling throughput. It is sometimes advantageous to fence off interrupts (isolate CPUs handling interrupts from CPUs running workload threads).

CPU Providers

The DTrace providers shown in Table 3-3 are used for examining CPU usage.

CPU One-Liners

You can use the following one-liners to get quick answers to important questions about what the CPUs are doing.

Table 3-3 Providers for Tracking CPU usage

Provider	Description
profile, tick	These providers allow for time-based data collection and are very useful for kernel and user CPU profiling.
sched	Observing scheduling activity is key to understanding CPU usage on loaded systems.
proc	The proc provider lets you observe key process/thread events.
sysinfo	This is important for tracking systemwide events that relate to CPU usage.
fbt	The function boundary tracing provider can be used to examine CPU usage by kernel function.
pid	The pid provider enables instrumenting unmodified user code for drilling down on application profiling.
lockstat	lockstat is both a DTrace consumer (`lockstat(1M)`) and a special provider used for observing kernel locks and kernel profiling.
syscall	Observing system calls is generally a good place to start, because system calls are where applications meet the kernel, and they can provide insight as to what the workload is doing.
plockstat	This provides statistics on user locks. It can identify lock contention in application code.

profile Provider

The profile provider samples activity across the CPUs; for these one-liners, a rate of 997 Hertz[1] is used to avoid sampling in lockstep with timed kernel tasks.

Which processes are on-CPU?

```
dtrace -n 'profile-997hz { @[pid, execname] = count(); }'
```

Which processes are on-CPU, running user code?

```
dtrace -n 'profile-997hz /arg1/ { @[pid, execname] = count(); }'
```

What are the top user functions running on-CPU (%usr time)?

```
dtrace -n 'profile-997hz /arg1/ { @[execname, ufunc(arg1)] = count(); }'
```

What are the top kernel functions running on-CPU (%sys time)?

```
dtrace -n 'profile-997hz /arg0/ { @[func(arg0)] = count(); }'
```

What are the top five kernel stack traces on the CPU (shows why)?

```
dtrace -n 'profile-997hz { @[stack()] = count(); } END { trunc(@, 5); }'
```

What are the top five user stack traces on the CPU (shows why)?

```
dtrace -n 'profile-997hz { @[ustack()] = count(); } END { trunc(@, 5); }'
```

What threads are on-CPU, counted by their thread name (FreeBSD)?

```
dtrace -n 'profile-997 { @[stringof(curthread->td_name)] = count(); }'
```

1. This means events per second.

sched Provider

Which processes are getting placed on-CPU (the sched provider, event-based)?

```
dtrace -n 'sched:::on-cpu { @[pid, execname] = count(); }'
```

Which processes are getting charged with CPU time when tick accounting is performed?

```
dtrace -n 'sched:::tick { @[stringof(args[1]->pr_fname)] = count(); }'
```

syscall Provider

What system calls are being executed by the CPUs?

```
dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
```

Which processes are executing the most system calls?

```
dtrace -n 'syscall:::entry { @[pid, execname] = count(); }'
```

What system calls are a given process name executing (for example, `firefox-bin`)?

```
dtrace -n 'syscall:::entry /execname == "firefox-bin"/ { @[probefunc] = count(); }'
```

CPU Analysis

Assuming the CPUs are not idle, a systemwide view of where the CPU cycles are going can be determined using the DTrace profile provider. This is a time-based provider; it does not instrument a specific area of code but rather allows the user to specify time intervals for the probes to fire. This makes it suitable for sampling what is occurring on the CPUs, which can be performed at a rate sufficiently high enough to give a reasonable view of what the CPUs are doing (for example, sampling at around 1000 Hertz).

We can start with a basic profile to determine which processes and threads are running on the CPUs to account for the user cycles and then look at a kernel profile to understand the sys cycles. In Solaris, the `prstat(1M)` utility is the easiest way to track which processes are the top consumers of CPU cycles, but here we'll start by taking a look at using DTrace to observe CPU usage.

This DTrace one-liner uses the profile provider to sample process IDs and process names that are on-CPU in user-mode code:

```
solaris# dtrace -n 'profile:::profile-997hz /arg1/ { @[pid, execname] = count(); }'
^C
[...output truncated...]
     2735  oracle.orig                                                       4088
     2580  oracle.orig                                                       4090
     2746  oracle.orig                                                       4093
     2652  oracle.orig                                                       4100
     2748  oracle.orig                                                       4108
     2822  oracle.orig                                                       4111
     2644  oracle.orig                                                       4112
     2660  oracle.orig                                                       4122
     2554  oracle.orig                                                       4123
     2668  oracle.orig                                                       4123
     2560  oracle.orig                                                       4131
     2826  oracle.orig                                                       4218
     2568  oracle.orig                                                       4229
     2836  oracle.orig                                                       4244
     2736  oracle.orig                                                       4277
     2654  oracle.orig                                                       4290
     2816  oracle.orig                                                       4320
     2814  oracle.orig                                                       4353
     2658  oracle.orig                                                       4380
     2674  oracle.orig                                                       7892
```

As introduced in Chapter 2, the profile provider has probes with the prefix `profile-`, which fire on all CPUs. The probe name includes the rate to sample for: Here 997 Hertz (997Hz) was specified. Avoiding rates of 1000 Hertz and 100 Hertz is a good idea when doing time-based collection to avoid sampling in lockstep with regular events such as the kernel clock interrupt.[2] It does not need to be 997Hz—you can sample less frequently (113Hz, 331Hz, 557Hz, and so on) if desired.

The profile probe has two arguments: `arg0` and `arg1`. `arg0` is the program counter (PC) of the current instruction if the CPU is running in the kernel, and `arg1` is the PC of the current instruction if the CPU is executing in user mode. Thus, the test `/arg0/ (arg0 != 0)` equates to "is the CPU executing in the kernel?" and `/arg1/ (arg1 != 0)` equates to "is the CPU executing in user mode?"

The profile one-liner included the predicate `/arg1/` to match on user-land execution. We see a pretty even distribution of different Oracle processes running on-CPU based on our frequent (997Hz, or just about every millisecond) sampling. We see

2. This happens every ten milliseconds (unless tuned) and is used to take care of some kernel housekeeping (statistics gathering, and so on).

process PID 2674 as showing up most frequently during the sample period. Let's take a quick look with a simple `ps(1)` command and see whether that makes sense:

```
solaris# ps -efcL | grep 2674
  oracle  2674     1     1    19   TS   0    May 27 ?          51:21 ora_lgwr_BTRW
  oracle  2674     1     2    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1     3    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1     4    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1     5    19   TS  46    May 27 ?          10:51 ora_lgwr_BTRW
  oracle  2674     1     6    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1     7    19   TS  19    May 27 ?          10:50 ora_lgwr_BTRW
  oracle  2674     1     8    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1     9    19   TS  50    May 27 ?          10:49 ora_lgwr_BTRW
  oracle  2674     1    10    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1    11    19   TS  41    May 27 ?          10:50 ora_lgwr_BTRW
  oracle  2674     1    12    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1    13    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1    14    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1    15    19   TS  59    May 27 ?           0:00 ora_lgwr_BTRW
  oracle  2674     1    16    19   TS  25    May 27 ?          10:48 ora_lgwr_BTRW
  oracle  2674     1    17    19   TS  60    May 27 ?          10:50 ora_lgwr_BTRW
  oracle  2674     1    18    19   TS  60    May 27 ?          10:50 ora_lgwr_BTRW
  oracle  2674     1    19    19   TS   3    May 27 ?          10:49 ora_lgwr_BTRW
```

Process PID 2674 is the Oracle database log writer, which has several busy threads. It is typical to see the log writer as a top CPU consumer in an Oracle workload, so this is not surprising. To continue understanding the workload and CPU usage in more detail, we can take a look at where the kernel is spending time (the SYS component of CPU utilization) using either `lockstat(1M)`, which uses the lockstat provider, or the profile provider.

```
solaris# dtrace -n 'profile-997hz /arg0 && curthread->t_pri != -1/
    { @[stack()] = count(); }
    tick-10sec { trunc(@, 10); printa(@); exit(0); }'
[...]
              FJSV,SPARC64-VII`copyout+0x468
              unix`current_thread+0x164
              genunix`uiomove+0x90
              genunix`struiocopyout+0x38
              genunix`kstrgetmsg+0x780
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
              genunix`read+0x274
              unix`syscall_trap+0xac
             1200

              FJSV,SPARC64-VII`cpu_smt_pause+0x4
              unix`current_thread+0x164
              platmod`plat_lock_delay+0x78
              unix`mutex_vector_enter+0x460
              genunix`cv_timedwait_sig_hires+0x1c0
              genunix`cv_waituntil_sig+0xb0
              semsys`semop+0x564
              unix`syscall_trap+0xac
             1208
```

```
              unix`disp_getwork+0xa0
              genunix`disp_lock_exit+0x58
              unix`disp+0x1b4
              unix`swtch+0x8c
              genunix`cv_wait_sig+0x114
              genunix`str_cv_wait+0x28
              genunix`strwaitq+0x238
              genunix`kstrgetmsg+0xdcc
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
              genunix`read+0x274
              unix`syscall_trap+0xac
             1757
```

Before we describe the output, we'll show the one-liner rewritten as a D script, making it easier to read and understand:

```
1   #!/usr/sbin/dtrace -s
2
3   profile-997hz
4   /arg0 && curthread->t_pri != -1/
5   {
6        @[stack()] = count();
7   }
8   tick-10sec
9   {
10       trunc(@,10);
11       printa(@);
12       exit(0);
13   }
```

Script kprof.d

The probe (line 3) is the profile provider, sampling at 997Hz. We make use of the DTrace logical AND operator to test for more than one condition in the predicate when the probe fires. Since we are interested only in the kernel this time, we're using a predicate that translates to "Are you running in the kernel?" (`arg0`, which equates to `arg0 != 0`) and (`curthread->t_pri != -1`), which is a Solaris-specific test to ensure that the CPU is not executing the kernel idle loop.

In Solaris, the idle loop priority will be set to -1 during execution. Table 3-4 shows equivalent predicates for Mac OS X and FreeBSD. These are all considered "unstable" since they refer to a curthread member and value, neither of which are public, stable interfaces; the Reference column in the table shows what these predicates are based on and should be double-checked before using these predicates in case there have been changes in your operating system version.

This script uses the tick probe to capture data for ten seconds, at which point the aggregation of kernel stacks collected when the profile probe fires will be truncated to the top ten (ten most frequent stack frames), the aggregation will be printed, and the script will exit.

Table 3-4 Predicates for Filtering Out the Idle Thread

OS	Predicate	Reference
Solaris	`/curthread->t_pri != -1/`	`thread_init()` in `usr/src/uts/common/disp/thread.c`
Mac OS X	`/!(curthread->state & 0x80)/`	`TH_IDLE` in `osfmk/kern/thread.h`
FreeBSD	`/!(curthread->td_flags & 0x20)/`	`TDF_IDLETD` in `/usr/src/sys/sys/proc.h`

Before getting into the specific example shown, here is a brief description of kernel and application profiling. Profiling refers to the process of determining in what area of code a piece of software is spending its time: what function or functions are executing most frequently. The profile is obtained using a sampling mechanism, where the program counter, which points to the currently executing instruction, is sampled at a predefined interval. The results of the profile are typically a list of software functions, sometimes in the form of stack frames, with either a count (indicating how many times a PC that resides in that function was captured in the sample) or a percentage (indicating what percent of time over the sampling period was spent in each of the functions). The `hotkernel` and `hotuser` scripts in the DTraceToolkit postprocess DTrace output and provide percentages.

How much sense you can make of the results of function profiling will depend on your experience, skill set, and knowledge of the software being profiled. But, even with minimal knowledge of the profile target, you'll still find it valuable to collect this information; it might be used by other parties involved in the problem diagnosis or the owners of the software.

Referring to the sample output, we show the top three kernel stack frames (in the interest of page space, we asked DTrace for the top ten). Once we get past the cryptic nature of stack frames, we can understand where the kernel is spending time. A given line in a kernel stack frame will include the name of the kernel module, followed by the backtick (`) character, followed by the name of the kernel function, and ending with the offset into the function (in hexadecimal) derived from the program counter. Here's an example of one line from the kernel stack frame, separating the three components to illustrate:

```
Kernel Module     Kernel Function     Offset into Kernel Function
sockfs            `socktpi_read       +0x44
```

Stack frames are read starting at the bottom, moving up to the top with each function call. Note that user stack frames have the same format: the module and function relating to the user process/thread being profiled and an offset showing the specific user PC.

The bottom stack frame, which was the most frequent frame during the sampling period, indicates the kernel was spending most of its time in reads of network sockets, and the second most frequent stack frame indicates that the kernel is handling semaphore system calls (which is not unusual for an Oracle workload). The third frame, at the top, is another kernel stack from network socket reads.

Another method for profiling the kernel is to use the `func()` function and `caller` variable in DTrace to generate output that can be easier to follow than paging through screenfuls of stack frames.

```
1   #!/usr/sbin/dtrace -s
2   #pragma D option quiet
3
4   profile-997hz
5   /arg0 && curthread->t_pri != -1/
6   {
7           @[func(caller), func(arg0)] = count();
8   }
9   tick-10sec
10  {
11          trunc(@,20);
12          printf("%-24s %-32s %-8s\n","CALLER","FUNCTION","COUNT");
13          printa("%-24a %-32a %-@8d\n",@);
14          exit(0);
15  }
```

Script kprof_func.d

The previous script uses several different keys in the count aggregation (line 7). First is `func(caller)`; this DTrace `func` function takes a PC as an argument and returns the symbolic name of the function the PC resides in. `caller` is a DTrace variable that contains the PC location of the current thread just before entering the current function. The first aggregation key provides the function that called the current function. It is essentially the top two entries off the stack frame. The script includes formatting in the `tick-10sec` probe, with a `printf()` statement to display headers that are left-justified and a `printa()` statement that formats the output in alignment with the headers. These DTrace features enable building scripts with formatted output, improving the readability of the generated data.

```
solaris# ./kprof_func.d
CALLER                   FUNCTION                         COUNT
unix`syscall_trap        genunix`sleepq_wakeall_chan      1607
```

continues

```
genunix`fop_rwlock        genunix`fop_rwlock               1652
unix`current_thread       unix`mutex_delay_default         1667
genunix`fop_rwunlock      genunix`fop_rwunlock             1691
ip`tcp_fuse_output        ip`tcp_fuse_output               1777
genunix`str_cv_wait       genunix`rwnext                   1797
unix`current_thread       ip`tcp_loopback_needs_ip         1927
0x28cdf48                 unix`disp_getwork                1962
unix`_resume_from_idle    unix`_resume_from_idle           2041
unix`current_thread       unix`lock_set                    2120
0x1400                    ip`tcp_fuse_output               2157
unix`current_thread       unix`lock_set_spl                2292
unix`current_thread       unix`mutex_exit                  2313
unix`current_thread       FJSV,SPARC64-VII`cpu_smt_pause   2656
unix`current_thread       unix`fp_restore                  2978
genunix`kstrgetmsg        genunix`kstrgetmsg               3102
0x0                       unix`utl0                        3782
unix`current_thread       FJSV,SPARC64-VII`copyout         4957
genunix`disp_lock_exit    unix`disp_getwork                5110
unix`current_thread       unix`mutex_enter                 17420
```

The output shows the top kernel function was `mutex_enter`, followed by `disp_getwork`, `copyout`, and so on. For details on what each of these kernel modules and functions do, see *Solaris Internals* (McDougall and Mauro, 2006) and the other texts in the bibliography.

If you want to take one more step on the kernel profile, obtaining a stack trace when the probe that corresponds to the top kernel function can be very informative, as shown in the next example:

```
solaris# dtrace -n 'fbt:unix:mutex_enter:entry'
dtrace: invalid probe specifier fbt:unix:mutex_enter:entry:
probe description fbt:unix:mutex_enter:entry does not match any probes

solaris# dtrace -l | grep mutex_enter
60103   lockstat           genunix                        mutex_enter adaptive-acquire
60104   lockstat           genunix                        mutex_enter adaptive-block
60105   lockstat           genunix                        mutex_enter adaptive-spin

solaris# dtrace -n 'lockstat:genunix:mutex_enter: { @[stack()] = count(); }'
[...]
              ip`tcp_fuse_rrw+0x14
              genunix`rwnext+0x254
              genunix`strget+0x8c
              genunix`kstrgetmsg+0x228
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
              genunix`read+0x274
              unix`syscall_trap+0xac
          1867400

              genunix`cv_wait_sig+0x13c
              genunix`str_cv_wait+0x28
              genunix`strwaitq+0x238
              genunix`kstrgetmsg+0xdcc
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
```

```
              genunix`read+0x274
              unix`syscall_trap+0xac
          1868297

              ip`squeue_enter+0x10
              sockfs`sostream_direct+0x194
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
          1893532
```

The previous example code shows three separate invocations of dtrace(1M) and is intended to illustrate another method of tracing software function flow. In the first invocation, we attempted to enable a probe for mutex_enter() using the fbt provider, and dtrace(1M) reported that no such probe exists. This may happen in rare cases when you attempt to enable a kernel function found in a stack frame; not every function in the kernel can be instrumented with the DTrace fbt provider.[3]

The next step was to determine whether probes do exist that correspond to the string mutex_enter, and we found that the lockstat provider manages three such probes. The third and final invocation instruments those three probes by leaving the fourth probe field, probename, blank, which instructs DTrace to do a wildcard match. The DTrace command will capture a kernel stack frame when the probes fire. The resulting output shows the top three kernel stack frames, and we can see that the mutex calls are the result of network reads and writes (write and read system calls entering the kernel sockfs module).

This again illustrates the drill-down methodology that DTrace facilitates: taking information provided during one phase of the investigation, a kernel function in this case, and creating a new DTrace invocation to further understand the source.

It is not uncommon for performance analysis or troubleshooting system behavior to require looking at the system and software from several angles, based on collected data and observations. We illustrate this here as we continue with our example, moving from profiling kernel time to taking an important component of that data (the system calls observed) and collecting relevant data to better understand that aspect of the load. Taking the next step with the data collected up to this point, we can track the source of the write and read system calls (bottom of the stack frames shown) and determine which processes are making the calls.

3. Reference Chapter 12 and the “fbt Provider” chapter in the DTrace Guide for more information on using the fbt provider and known limitations on instrumenting some functions.

```
solaris# dtrace -n 'syscall::read:entry,syscall::write:entry
/fds[arg0].fi_fs == "sockfs"/ { @[execname,pid] = count(); }'
[...]
  oracle.orig                                              8868            57888
  oracle.orig                                              8772            57892
  rwdoit                                                   8724            57894
  rwdoit                                                   8914            57918
  oracle.orig                                              8956            57920
  oracle.orig                                              8872            58434
  rwdoit                                                   8849            58434
  oracle.orig                                              9030            58511
  rwdoit                                                   8982            58512
  oracle.orig                                              8862            58770
  rwdoit                                                   8816            58772
  oracle.orig                                              8884            59068
  rwdoit                                                   8846            59068
  oracle.orig                                              8778            59616
  rwdoit                                                   8735            59616
  rwdoit                                                   8909            62624
  oracle.orig                                              8954            62626
  rwdoit                                                   8844            62776
  oracle.orig                                              8874            62778
```

The DTrace program executed on the command line shown previously enables two probes: the entry points for the read and write system calls. The predicate uses the DTrace `fds[]` array variable for file description information. We use `arg0` to index the `fds[]` array, which is the file descriptor passed to both the read and write system calls. Among the file data made available in the `fds[]` array is the file system type associated with the file descriptor. We know from the stack frame that the reads and writes are on sockets, which is a special file type that represents a network endpoint. Thus, in the predicate, we're instructing DTrace to take action only on reads and writes to network sockets.

The action in the DTrace program is another use of the count aggregation, keyed on the process name (`execname`) and PID. The resulting output (`execname`, PID, and the count in the rightmost column) shows that basically all our Oracle workload processes are generating the network I/Os, which accounts for why we see the kernel spending most of its time in network code.

Continuing to drill down and again illustrating how much can be learned about precisely what your system is doing, we can determine how many bytes per second are being read and written over the network connections.

```
solaris# dtrace -qn 'syscall::read:entry,syscall::write:entry
/fds[arg0].fi_fs == "sockfs"/ { @[probefunc] = sum(arg2); }
tick-1sec { printa(@); trunc(@); }'

  write                                                      34141396
  read                                                     1832682240

  write                                                      33994014
  read                                                     1822898304
```

```
write                                                            33950736
read                                                           1824884640

write                                                            33877395
read                                                           1820879136
```

We modified our DTrace program on the command line. Using the same probes and predicate as in the previous example, we changed the action taken when the probe fires (we also added the `-q` flag to the `dtrace` command, to enable quiet mode). In the action, we now use the `sum()` function to maintain an arithmetic sum of the passed value, in this case `arg2`, which for read and write system calls is the number of bytes to read or write with the call (reference the man page for a system call of interest to determine what arguments are passed). The resulting output indicates that the system is writing about 34MB/sec and reading 1.8 GB/sec. Note that system calls such as read and write may not actually read or write the number of bytes requested. The return value from read and write provides the actual number of bytes, and this can be instrumented using the return probes for those system calls, shown here as a script:

```
1  #! /usr/sbin/dtrace -qs
2
3  syscall::read:entry,syscall::write:entry
4  /fds[arg0].fi_fs == "sockfs"/
5  {
6       self->flag = 1
7  }
8  syscall::read:return,syscall::write:return
9  /(int)arg0 != -1 && self->flag/
10 {
11      @[probefunc] = sum(arg0);
12 }
13 syscall::read:return,syscall::write:return
14 {
15      self->flag = 0;
16 }
```

Script rw_bytes.d

The `rw_bytes.d` script sets a flag at the entry of the system calls (line 6), which is tested in the predicate for the return probes (line 9). `arg0` in the return probes is the value returned by the system call, which for read and write is the number of bytes actually read or written, or -1 if there was an error. The predicate filters out the error state so that -1 is not added to the `sum()` by accident.

Note also that the script instruments only the read and write systems calls (as does the previous command-line example). Variants on those calls, such as `pread(2)`, `pread64(2)`, `read_nocancel(2)`, and so on, will not be instrumented. That can be accomplished to some degree by using the asterisk (`*`) pattern-matching character in the probe name, for example, `syscall::*read*:entry`. Consult the man pages on the system calls that will be matched in this way to ensure that `arg0` is used as expected for all instrumented calls, because it is used in the `rw_bytes.d` script. Also, on Mac OS X, `syscall::*read*:entry` will match several system calls that are not related to I/O at all:

```
macosx> dtrace -ln 'syscall::*read*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
18506    syscall                                                read entry
18616    syscall                                            readlink entry
18740    syscall                                               readv entry
18806    syscall                                               pread entry
19136    syscall                                            aio_read entry
19156    syscall                                      __pthread_kill entry
19158    syscall                                   __pthread_sigmask entry
19162    syscall                                __disable_threadsignal entry
19164    syscall                                  __pthread_markcancel entry
19166    syscall                                    __pthread_canceled entry
19196    syscall                                     __pthread_chdir entry
19198    syscall                                    __pthread_fchdir entry
19220    syscall                                    bsdthread_create entry
19222    syscall                                 bsdthread_terminate entry
19232    syscall                                  bsdthread_register entry
19244    syscall                                       thread_selfid entry
19292    syscall                                       read_nocancel entry
19322    syscall                                      readv_nocancel entry
19328    syscall                                      pread_nocancel entry
```

Having looked at the system call dimension of the load, we will now turn our attention back to the %sys (kernel) component of CPU utilization.

Another approach to understanding kernel CPU utilization is to use the fbt provider, which enables instrumenting the entry and return of most functions in the kernel. Note that fbt manages a great many probes, so enabling a large number of fbt probes on a busy system spending time in the kernel will potentially have a noticeable probe effect.

As an example, let's get a broad view of which functions are being executed most frequently by the kernel, by first tracking which kernel modules show up in a count aggregation.

```
solaris# dtrace -n 'fbt:::entry { @k[probemod] = count(); }'
dtrace: description 'fbt:::entry ' matched 29021 probes
^C
[...]
  emlxs                                                          432236
  tmpfs                                                         3552878
  TS                                                            5364110
  FJSV,SPARC64-VII                                              5403467
  platmod                                                      10341211
  sockfs                                                       20297174
  ip                                                           20526931
  unix                                                        117374161
  genunix                                                     312204419
```

The D program in the previous example enables a probe at the entry point of every kernel function (29,021 probes). The count aggregation data shows calls in the `genunix` and `unix` modules were the most frequent, followed by `ip` and `sockfs`, so significant networking activity is taking place. We can drill down further by honing in on just the `genunix` module:

```
solaris# dtrace -n 'fbt:genunix::entry { @k[probefunc] = count(); }'
dtrace: description 'fbt:genunix::entry ' matched 6267 probes
^C
[...]
  times                                                        3018463
  disp_lock_exit_high                                          3353612
  cv_broadcast                                                 3882154
  mstate_aggr_state                                            6036982
  syscall_mstate                                               9838849
```

We added a module name to the DTrace probe (`genunix`) and changed the aggregation key from `probemod` (which gave us kernel modules) to `probefunc`, which will provide kernel function names in the `genunix` kernel module. Note the top kernel functions displayed, `syscall_mstate()` and `mstate_aggr_state()`. We can drill down one more level to get the complete picture:

```
solaris# dtrace -n 'fbt:genunix:syscall_mstate:entry { @k[stack()] = count(); }'
dtrace: description 'fbt:genunix:syscall_mstate:entry ' matched 1 probe
^C
[...]
              unix`syscall_trap+0x114
         10467759

              unix`syscall_trap+0x68
         10493467
```

Here we gathered kernel stack frames when `syscall_mstate()` was called, and we see it is being called right out of the system call trap handler. So, the calls from functions in the genunix module are managing a high rate of system calls and calling into the associated per-thread microstate accounting (mstate) code. We can apply the same set of steps to the `unix` kernel module, which we saw in the fbt kernel module count.

```
solaris# dtrace -n 'fbt:unix::entry { @k[probefunc] = count(); }'
dtrace: description 'fbt:unix::entry ' matched 2179 probes
^C
[...]
  bitset_in_set                                                1867698
  disp_getwork                                                 2331845
  default_lock_backoff                                         2915223
```

continues

```
  cmt_ev_thread_swtch                                         5092075
  bitset_find_in_word                                         5244363

solaris# dtrace -n 'fbt:unix:bitset_find_in_word:entry { @k[stack()] = count(); }'
dtrace: description 'fbt:unix:bitset_find_in_word:entry ' matched 1 probe
^C
[...]
              unix`bitset_find+0x64
              unix`cpu_wakeup+0x80
              genunix`sleepq_wakeall_chan+0x48
              genunix`cv_broadcast+0x4c
              ip`tcp_fuse_output+0x7f0
              ip`tcp_output+0x74
              ip`squeue_drain+0x130
              ip`squeue_enter+0x348
              sockfs`sostream_direct+0x194
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
           997431

              unix`bitset_find+0x64
              unix`cpu_wakeup+0x80
              genunix`sleepq_wakeall_chan+0x48
              genunix`cv_broadcast+0x4c
              ip`tcp_fuse_output+0x7f0
              ip`tcp_output+0x74
              ip`squeue_enter+0x74
              sockfs`sostream_direct+0x194
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
          6778835
```

The previous example uses the same set of steps shown previously. First we gathered a kernel function call count for the `unix` module, followed by gathering kernel stack frames on the entry point of the most frequent kernel function (`bitset_find_in_word()`) from that module. We can see from the stack frame that the `bitset_find_in_word()` call is originating from a high volume of write system calls to network sockets and the kernel issuing a wake-up to sleeping threads.

Before moving on, we should point out again that use of the fbt provider is an advanced use of DTrace; proper interpretation of the data requires knowledge of the kernel, and you may need to examine the kernel source code to understand what a particular module or function does. The fbt provider manages a lot of probes, so use it with care on busy systems spending time in the kernel and potentially enabling a large number of fbt probes.

Kernel profiling applies of course when CPUs are spending enough time in the kernel to warrant having a look. If your system CPU utilization shows most of the CPU cycles are in user mode, it's useful to determine what is running and profile the user processes. Recall that the CPU utilization for this example indicated about 60 percent of CPU time is spent in user mode. Here's a sample using the Solaris `mpstat(1M)` command:

```
solaris# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 3660  1678   38 5715  101 2946  124    6 20093   58  33   0   9
  1    0   0 3782  1797   58 5998  108 2928  129   11 21123   59  29   0  12
  2    0   0 3632  1695   57 5968  111 3101  125   11 20523   62  29   0   9
  3    0   0 3579  1850   53 5832  104 2898  127    6 20498   61  28   0  11
  4    0   0 3525  1697   47 5781   82 3005  122    9 20221   63  28   0   9
  5    0   0 3620  1987  350 5375   79 2746  142    6 19332   57  33   0  10
  6    0   0 3644  1707   51 5821   95 3013  121    4 19564   62  29   0   9
  7    0   0 3529  1809   47 5725   86 2893  112    6 19878   61  28   0  11
  8    0   0 3725  1862   62 6029  103 3124  159    9 20713   61  29   0  10
. . .
```

We know from previous examples that we have many Oracle processes running on the CPUs. In Solaris, nonetheless, it can be useful to capture a few samples with `prstat(1M)` to determine which processes are the top CPU consumers. On non-Solaris platforms, `top(1)` can be used to accomplish the same task.

```
solaris# prstat -c 1
  PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  2674 oracle     38G   38G sleep    0    0   6:03:27 1.1% oracle.orig/19
  9568 oracle     38G   38G sleep    0    0   0:03:58 0.8% oracle.orig/1
  9566 oracle     38G   38G cpu24    0    0   0:03:57 0.8% oracle.orig/1
  9570 oracle     38G   38G sleep    0    0   0:03:57 0.8% oracle.orig/1
  9502 oracle     38G   38G cpu19    0    0   0:03:57 0.8% oracle.orig/1
  9762 oracle     38G   38G cpu12    0    0   0:03:58 0.8% oracle.orig/1
  9500 oracle     38G   38G sleep    0    0   0:03:57 0.8% oracle.orig/1
  9736 oracle     38G   38G cpu18    0    0   0:03:50 0.8% oracle.orig/1
  9580 oracle     38G   38G sleep    0    0   0:03:50 0.8% oracle.orig/1
  9662 oracle     38G   38G cpu28    0    0   0:03:48 0.8% oracle.orig/1
  9508 oracle     38G   38G cpu13    0    0   0:03:48 0.8% oracle.orig/1
  9734 oracle     38G   38G cpu31    0    0   0:03:46 0.8% oracle.orig/1
  9562 oracle     38G   38G sleep    0    0   0:03:44 0.8% oracle.orig/1
  9668 oracle     38G   38G sleep    0    0   0:03:44 0.8% oracle.orig/1
  9748 oracle     38G   38G sleep    0    0   0:03:43 0.8% oracle.orig/1
Total: 281 processes, 1572 lwps, load averages: 57.01, 51.62, 51.27
```

`prstat(1)` provides another view of an Oracle workload, with many Oracle shadow processes consuming CPU cycles. `prstat(1)` also shows us which processes have more than one thread (the number value following the / in the last column). We see process PID 2674 has 19 threads. We can use another `prstat(1)` invocation to take a closer look just at that process and where the individual threads are spending time:

```
# prstat -cLmp 2674
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  2674 oracle    14 8.7 0.0 0.0 0.0 0.0  76 1.8  4K 130 10K   0 oracle.orig/1
  2674 oracle   0.4 4.7 0.0 0.0 0.0  86 8.8 0.5  1K 637  1K   0 oracle.orig/18
  2674 oracle   0.4 4.6 0.0 0.0 0.0  85 9.1 0.5  1K 624  1K   0 oracle.orig/5
  2674 oracle   0.4 4.6 0.0 0.0 0.0  85 9.4 0.5  1K 609  1K   0 oracle.orig/19
  2674 oracle   0.4 4.6 0.0 0.0 0.0  86 8.7 0.4  1K 632  1K   0 oracle.orig/16
```
continues

```
   2674 oracle   0.4 4.6 0.0 0.0 0.0  85 9.2 0.5  1K 624  1K   0 oracle.orig/9
   2674 oracle   0.4 4.6 0.0 0.0 0.0  85 9.3 0.5  1K 610  1K   0 oracle.orig/7
   2674 oracle   0.4 4.5 0.0 0.0 0.0  85 9.3 0.5  1K 613  1K   0 oracle.orig/17
   2674 oracle   0.4 4.4 0.0 0.0 0.0  86 9.2 0.5  1K 585  1K   0 oracle.orig/11
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/15
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/14
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/13
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/12
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/10
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/8
 Total: 1 processes, 19 lwps, load averages: 56.07, 56.48, 55.46
```

Refer to the `prstat(1)` man page for details on the command line and what the individual columns mean. Basically, we asked `prstat(1)` to display the percentage of time the individual threads are spending in each microstate (columns `USR` through `LAT` are thread microstates tracked in the kernel).

We can take a closer look at the user component of CPU utilization using several methods. Earlier, we demonstrated a couple of one-liners that show which processes are getting on-CPU, using both the DTrace sched provider and the profile provider. Let's take another quick look:

```
solaris# dtrace -n 'sched:::on-cpu { @[execname, pid] = count(); }'
[...]
  oracle.orig                                           10194        16124
  oracle.orig                                           10362        16149
  oracle.orig                                           10370        16186
  oracle.orig                                           10268        16231
  oracle.orig                                           10290        16249
  oracle.orig                                           10352        16258
  rwdoit                                                10149        16260
  oracle.orig                                           10444        16296
  rwdoit                                                10245        16329
  rwdoit                                                10415        16401
  oracle.orig                                           10192        16967
  oracle.orig                                           10280        17035
  oracle.orig                                           10456        17114
  oracle.orig                                            2674        35566
  sched                                                     0      1361419
```

Again, the top user process is the Oracle log writer, PID 2674.

The variables in the aggregation key are printed in order. In this case, starting from the left, the output is the process name (`execname`), PID, and the last column on the right is the count. The data aligns with the `prstat(1M)` output, showing processes PID 2674 as the top user process. The largest count item is `sched`, which is the Solaris kernel (`sched` is the PID 0 process name). Mac OS X uses `kernel_task`.

We can explore this further by using `/execname == "sched"/` as a predicate with the `sched:::on-cpu` probe and capturing a stack trace:

```
# dtrace -n 'sched:::on-cpu /execname == "sched"/ { @[stack()] = count(); }'
dtrace: description 'sched:::on-cpu ' matched 3 probes
^C
[...]

              unix`_resume_from_idle+0x228
              unix`idle+0xb4
              unix`thread_start+0x4
          1020020

              unix`_resume_from_idle+0x228
              unix`idle+0x140
              unix`thread_start+0x4
          1739939
```

As we can see, there is nothing of much interest here; we're in the kernel idle loop for the most part when `sched` is on-CPU.

Let's take another look at the user mode component of the load. Drilling down further into the kernel component, while potentially interesting, is not essential for this example because we're not chasing CPU cycles being consumed in the kernel; we know that it's user processes burning CPU time. But we wanted to illustrate how to use DTrace to drill down further on observed data and also show that there may be situations where digging into kernel source code helps to further understand the source of specific events.

Moving on to profiling the user component of our sample, we observed processes called `oracle.orig` dominating the on-CPU profile. There are several different ways we can take a closer look at these processes. First, let's find out what system calls the busy user processes are executing:

```
solaris# dtrace -n 'syscall:::entry /execname == "oracle.orig"/
{ @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 234 probes
^C

  sysconfig                                                        2
  mmap                                                             8
  getloadavg                                                      10
  pollsys                                                         34
  pread                                                           38
  ioctl                                                          136
  close                                                          222
  open                                                           222
  lwp_sigmask                                                    774
  sigaction                                                     2322
  nanosleep                                                     3479
  yield                                                         7824
  pwrite                                                       33957
  kaio                                                         66733
  lwp_park                                                     67914
  semsys                                                      192263
  write                                                      3848880
  read                                                       3849103
  times                                                     27201971
```

Here we see the `times(2)` system call is the most frequently executed, followed by `read(2)`, `write(2)`, and `semsys(2)`—all very typical for an Oracle workload. A simple change to the predicate, and we get the same information for another process that appeared in our earlier profile, `rwdoit`:

```
solaris# dtrace -n 'syscall:::entry /execname == "rwdoit"/
{ @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 234 probes
^C

  gtime                                                          912832
  read                                                          2574616
  write                                                         2574621
```

Here we see the `rwdoit` process is all about `write(2)` and `read(2)` system calls. Again, we can look at what type of files are being read and written by these processes:

```
solaris# dtrace -n 'syscall::read:entry,syscall::write:entry
/execname == "rwdoit"/ { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::read:entry,syscall::write:entry ' matched 2 probes
^C

  sockfs                                                        5054183
```

When looking at I/O targets, it can be useful to start by determining the file system type, which in turn can make it easier to fine-tune the next DTrace program for a closer look. In this case, all the reads and writes are to the socket file system (`sockfs`), which means it's all network I/O (which we observed earlier).

Getting back to the `oracle.orig` processes, it is often useful to obtain a user stack leading up to frequently called system calls. This may or may not offer much insight, depending on your familiarity with the code. Nonetheless, obtaining this information and passing it on to a development team enables them to more quickly determine whether there is an opportunity for improvement (or even a bug).

```
solaris# dtrace -qn 'syscall::times:entry /execname == "oracle.orig"/
{ @[ustack()] = count(); } END { trunc(@, 2); exit(0); }'
^C

              libc.so.1`times+0x4
              oracle.orig`kews_cln_timestate+0x80
              oracle.orig`ksudlc+0x9b0
              oracle.orig`kssdel+0xc0
              oracle.orig`ksupop+0x8e4
              oracle.orig`opiodr+0x724
              oracle.orig`ttcpip+0x420
```

```
              oracle.orig`opitsk+0x5e8
              oracle.orig`opiino+0x3e8
              oracle.orig`opiodr+0x590
              oracle.orig`opidrv+0x448
              oracle.orig`sou2o+0x5c
              oracle.orig`opimai_real+0x130
              oracle.orig`ssthrdmain+0xf0
              oracle.orig`main+0x134
              oracle.orig`_start+0x17c
            29356

              libc.so.1`times+0x4
              oracle.orig`ksupucg+0x5d8
              oracle.orig`opiodr+0x358
              oracle.orig`ttcpip+0x420
              oracle.orig`opitsk+0x5e8
              oracle.orig`opiino+0x3e8
              oracle.orig`opiodr+0x590
              oracle.orig`opidrv+0x448
              oracle.orig`sou2o+0x5c
              oracle.orig`opimai_real+0x130
              oracle.orig`ssthrdmain+0xf0
              oracle.orig`main+0x134
              oracle.orig`_start+0x17c
            29357
```

The previous example truncated the aggregation to just the top two user stack frames. This is generally a good idea on large, busy machines running large, enterprise workloads, because the sheer number of processes and unique stack frames can be extremely large and take a very long time to process once the DTrace invocation is terminated. Note that the stack frames represent the function call path through user code, in this case the `oracle.orig` executable. The function names may or may not provide the DTrace user insight as to what the code is doing leading up to the execution of the system call. As we stated earlier, the software company or organization that owns the code is best equipped to make sense of the stack frames, because they will have access to, and knowledge of, the source code.

It is often useful to examine the workload process that is the top CPU consumer. Recall from previous examples that the Oracle logwriter process is our top CPU user in this example. In Solaris, using `prstat(1)` is the best place to start to profile the time of a specific process. We showed an example of this earlier. Here's another sample:

```
# prstat -Lmp 2674 -c 1
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  2674 oracle    16 9.3 0.0 0.0 0.0 0.0  73 1.4  1K  31  2K   0 oracle.orig/1
  2674 oracle   0.4 5.1 0.0 0.0 0.0  86 8.1 0.5 304 164 390   0 oracle.orig/11
  2674 oracle   0.4 5.0 0.0 0.0 0.0  87 7.5 0.4 301 150 389   0 oracle.orig/7
  2674 oracle   0.4 5.0 0.0 0.0 0.0  87 7.5 0.6 311 152 388   0 oracle.orig/17
  2674 oracle   0.4 5.0 0.0 0.0 0.0  86 7.7 0.5 310 152 391   0 oracle.orig/19
  2674 oracle   0.4 4.9 0.0 0.0 0.0  86 8.1 0.5 305 154 393   0 oracle.orig/9
```
continues

```
   2674 oracle   0.4 4.8 0.0 0.0 0.0  87 7.5 0.5 306 147 392   0 oracle.orig/18
   2674 oracle   0.4 4.7 0.0 0.0 0.0  87 7.8 0.5 314 146 387   0 oracle.orig/16
   2674 oracle   0.4 4.7 0.0 0.0 0.0  87 7.7 0.5 305 143 392   0 oracle.orig/5
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/15
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/14
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/13
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/12
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/10
   2674 oracle   0.0 0.0 0.0 0.0 0.0 100 0.0 0.0   0   0   0   0 oracle.orig/8
Total: 1 processes, 19 lwps, load averages: 57.39, 57.11, 54.96
```

The `prstat(1)` data shows the logwriter process has one relatively busy thread (thread 1), while the other threads spend most of their time waiting on a user lock (`LCK`), which is most likely a user-defined mutex lock associated with a condition variable. The microstate profile we get from `prstat` shows the busy thread spending 16 percent of time running on-CPU in user mode and 9.3 percent of time running on-CPU in the kernel (`USR` and `SYS` columns). We can get a time-based sample of which user functions the process is spending time in using the profile provider:

```
solaris# dtrace -n 'profile-1001hz /arg1 && pid == 2674/ { @[ufunc(arg1)] =
count(); }'
dtrace: description 'profile-1001hz ' matched 1 probe
^C
[...]
  oracle.orig`skgfospo                                          126
  oracle.orig`ksbcti                                            130
  oracle.orig`ksl_postm_add                                     137
  oracle.orig`dbgtTrcData_int                                   143
  oracle.orig`ksfd_update_iostatsbytes                          146
  libc.so.1`clear_lockbyte                                      175
  oracle.orig`kslgetl                                           201
  libc.so.1`mutex_lock_impl                                     237
  oracle.orig`kcrfw_post                                        278
  oracle.orig`_$c1A.kslpstevent                                 355
  oracle.orig`kcrfw_redo_write                                  638
```

As when we examined user stack frames, the output here is a list of user functions—the executable object where the function resides, followed by the backtick (`) character, followed by the actual function name. The column on the right is the count value from the `count()` aggregating function we used in the D program. The most frequent function is `kcrfw_redo_write()`, which we can infer initiates a write to the Oracle redo log file (pretty much what we would expect examining the Oracle log writer process).

We can also see how much time the threads are spending on-CPU once they get scheduled, using the following DTrace script:

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4  inline int MAX = 10;
5
6  dtrace:::BEGIN
7  {
8          start = timestamp;
9          printf(“Tracing for %d seconds...hit Ctrl-C to terminate sooner\n”, MAX);
10 }
11
12 sched:::on-cpu
13 /pid == $target/
14 {
15         self->ts = timestamp;
16 }
17
18 sched:::off-cpu
19 /self->ts/
20 {
21         @[cpu] = sum(timestamp - self->ts);
22         self->ts = 0;
23 }
24
25 profile:::tick-1sec
26 /++x == MAX/
27 {
28         exit(0);
29 }
30
31 dtrace:::END
32 {
33         printf("\nCPU distribution over %d milliseconds:\n\n",
34             (timestamp - start) / 1000000);
35         printf("CPU microseconds\n--- ------------\n");
36         normalize(@, 1000);
37         printa("%3d %@d\n", @);
38 }
```

Script wrun.d

The `wrun.d` script is a slightly modified version of `/usr/demo/dtrace/whererun.d`. The differences are the predicate for the on-CPU probe that matches the specified process and changing the maximum seconds to the integer variable `MAX`, which is printed in the `BEGIN` action. This script will exit automatically after ten seconds (which can be modified on line 4), and it tracks the total time a process was on a particular CPU over that ten-second period.

```
# ./wrun.d -p 2674
Tracing for 10 seconds...hit Ctrl-C to terminate sooner

CPU distribution over 10000 milliseconds:

CPU microseconds
--- ------------
 33 204
 61 1041
```

continues

```
   56 1220
   57 1326
   50 1339
   58 1364
   54 1600
   48 1658
   42 1702
   51 1743
   45 1815
   36 1817
   47 1940
   46 2042
   41 2120
   49 2293
   62 2314
   60 2314
   43 2353
   39 2436
   37 2478
   38 2486
   44 2566
   55 2731
   34 2767
   35 2817
   40 2958
   53 3190
   63 3515
   59 3659
   52 4256
   32 4285
   20 154417
   21 155361
   15 161505
    8 163002
   31 163594
    1 164015
    9 165568
   29 166454
   19 166781
   18 166828
   11 166909
   27 168510
    6 169076
    7 169084
   14 169555
   28 172845
   26 175216
    4 175753
   10 176914
    5 179316
    0 180847
   16 180906
   30 181307
   25 183428
   22 186952
    2 189294
    3 189756
   12 190448
   17 192988
   23 193152
   13 196606
   24 199634
```

The output from the `wrun.d` script shows the time in nanoseconds the various threads in the logwriter process spent on the available system CPUs. The data indicates that the threads in this process are not very CPU-bound (which we also observed in the `prstat(1)` data), with a maximum time of 199,634 nanoseconds (about 200 microseconds) spent on-CPU 24 over ten seconds of wall clock time.

Having looked at CPU usage with some profiles and utilization scripts, along with user processes getting on-CPU, let's take a look at run queue latency. From a performance perspective, it is very useful to know whether runnable threads are spending an inordinate amount of time waiting for their turn on a CPU and whether some CPUs have longer wait times than others. We can use the sched provider for this:

```
1   #!/usr/sbin/dtrace -s
2   #pragma D option quiet
3   sched:::enqueue
4   {
5           s[args[0]->pr_lwpid, args[1]->pr_pid] = timestamp;
6   }
7
8   sched:::dequeue
9   /this->start = s[args[0]->pr_lwpid, args[1]->pr_pid]/
10  {
11          this->time = timestamp - this->start;
12          @lat_avg[args[2]->cpu_id] = avg(this->time);
13          @lat_max[args[2]->cpu_id] = max(this->time);
14          @lat_min[args[2]->cpu_id] = min(this->time);
15          s[args[0]->pr_lwpid, args[1]->pr_pid] = 0;
16
17  }
18  tick-1sec
19  {
20          printf("%-8s %-12s %-12s %-12s\n", "CPU", "AVG(ns)",
 "MAX(ns)", "MIN(ns)");
21          printa("%-8d %-@12d %-@12d %-@12d\n", @lat_avg, @lat_max, @lat_min);
22          trunc(@lat_avg); trunc(@lat_max); trunc(@lat_min);
23  }
```

Script lat.d

This script uses the DTrace `avg()`, `max()`, and `min()` functions to collect the data for column output, which is printed in a per-second tick probe. We also demonstrate the use of the `printa()` function to format the data and display multiple aggregations (line 24).

The `sched:::enqueue` probe fires when a thread is placed on a CPU run queue, where a time stamp is stored in the global variable `s` (line 5), which is an associative array, indexed by the `lwpid` and process PID derived from the arguments available to that probe. When the dequeue probe fires, a thread is being dequeued to be placed on a CPU. The predicate (line 9) both assigns the array value to `this->start` and checks that it is not zero, which ensures that the enqueue event was traced and the start time is known.

```
CPU       AVG(ns)       MAX(ns)       MIN(ns)
[...]
24        45050         1532700       6100
22        45138         424700        4800
14        45144         507500        5500
21        45200         478200        4000
26        45215         529100        5000
17        45269         462900        4500
5         45311         832800        4700
4         45381         370800        5100
3         45414         697300        4700
20        45465         383200        7100
16        45521         675700        5300
12        45709         656300        5400
29        46143         438500        5000
23        46346         378000        4400
15        46460         376300        5000
6         46484         500600        4500
13        46534         506300        5400
28        46561         338400        5300
18        46577         486900        6500
10        46665         450000        4700
30        46982         837800        4300
31        47593         394700        5700
0         48973         661800        7200
7         60798         52566200      3800
```

The output shown previously (truncated for space) shows the average run queue latency time runs in the 45-microsecond to 60-microsecond range, with the low end in the 5-microsecond to 6-microsecond range and a couple max values in the milliseconds. CPU 7 had the largest wait of 52 milliseconds.

For a per process/thread view, the Solaris `prstat(1)` command, with microstates and per-thread statistics, is a great tool to observe what percentage of time threads are runnable, waiting for a CPU.

```
solaris# prstat -cLm 1
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
 25888 oracle    40  13 0.0 0.0 0.0 0.0  42 4.6  2K  40 17K   0 oracle.orig/1
 25812 oracle    39  13 0.0 0.0 0.0 0.0  43 4.8  2K  26 17K   0 oracle.orig/1
 25806 oracle    39  13 0.0 0.0 0.0 0.0  43 4.6  2K  37 17K   0 oracle.orig/1
 26066 oracle    39  13 0.0 0.0 0.0 0.0  44 4.6  1K  38 17K   0 oracle.orig/1
 25820 oracle    39  13 0.0 0.0 0.0 0.0  44 4.6  1K  44 17K   0 oracle.orig/1
 26076 oracle    38  13 0.0 0.0 0.0 0.0  44 4.6  1K  39 16K   0 oracle.orig/1
 25986 oracle    38  12 0.0 0.0 0.0 0.0  45 4.3  1K  38 16K   0 oracle.orig/1
 25882 oracle    38  12 0.0 0.0 0.0 0.0  45 4.3  1K  41 16K   0 oracle.orig/1
 26080 oracle    38  13 0.0 0.0 0.0 0.0  45 4.6  1K  33 16K   0 oracle.orig/1
 25808 oracle    38  12 0.0 0.0 0.0 0.0  45 4.5  1K  30 16K   0 oracle.orig/1
 25892 oracle    37  13 0.1 0.0 0.0 0.0  45 4.4  1K  41 16K   0 oracle.orig/1
 25810 oracle    38  13 0.0 0.0 0.0 0.0  45 4.7  1K  45 16K   0 oracle.orig/1
 25804 oracle    38  12 0.0 0.0 0.0 0.0  45 4.5  1K  39 16K   0 oracle.orig/1
 26082 oracle    39  11 0.0 0.0 0.0 0.0  46 4.0  2K  34 17K   0 oracle.orig/1
 25980 oracle    38  12 0.0 0.0 0.0 0.0  48 2.4  2K  18 20K   0 oracle.orig/1
Total: 277 processes, 1569 lwps, load averages: 57.53, 56.45, 54.35
```

The LAT column is defined as run queue latency. In this sample, we see our Oracle processes spending about 4.6 percent of their time in each one-second sampling interval waiting for a CPU. Using DTrace, we can measure the actual time spent on the run queue for an individual process (or thread).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
sched:::enqueue
/args[1]->pr_pid == $target/
{
        s[args[2]->cpu_id] = timestamp;
}

sched:::dequeue
/s[args[2]->cpu_id]/
{
        @lat_sum[args[1]->pr_pid] = sum(timestamp - s[args[2]->cpu_id]);
        s[args[2]->cpu_id] = 0;
}

tick-1sec
{
        normalize(@lat_sum, 1000);
        printa("PROCESS: %d spent %@d microseconds waiting for a CPU\n", @lat_sum);
        trunc(@lat_sum);
}
```

Script plat.d

The plat.d script uses a predicate with the sched:::enqueue probe that checks the PID of the process is enqueued with the $target DTrace macro, which is expanded to the PID passed on the command line when the script is executed. The sum function is used to track the total time spent between enqueue and dequeue every second.

```
solaris# ./plat.d -p 2674

PROCESS: 2674 spent 119481 microseconds waiting for a CPU
PROCESS: 2674 spent 122856 microseconds waiting for a CPU
PROCESS: 2674 spent 134672 microseconds waiting for a CPU
PROCESS: 2674 spent 125041 microseconds waiting for a CPU
PROCESS: 2674 spent 117196 microseconds waiting for a CPU
PROCESS: 2674 spent 120019 microseconds waiting for a CPU
PROCESS: 2674 spent 117465 microseconds waiting for a CPU
PROCESS: 2674 spent 118093 microseconds waiting for a CPU
PROCESS: 2674 spent 118528 microseconds waiting for a CPU
^C
```

The resulting output shows that our target process spends about 110 milliseconds in each one-second period waiting on a run queue.

It can also be useful to track CPU run queue depth across all the CPUs on the system. Run queue depth refers to the number of runnable threads sitting on run queues waiting for their turns on a CPU. The following script is from the DTrace Guide:

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  sched:::enqueue
6  {
7          this->len = qlen[args[2]->cpu_id]++;
8          @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
9  }
10
11 sched:::dequeue
12 /qlen[args[2]->cpu_id]/
13 {
14         qlen[args[2]->cpu_id]--;
15 }
```

Script rq.d

Using the sched provider's enqueue and dequeue probes (lines 5 and 11), we increment an array variable, `qlen`, indexed by the CPU ID, and set a clause-local variable (`this->len`) to that value (line 7). The aggregation on line 8 uses `lquantize()` to provide a linear plot of per-CPU run queue depth. The dequeue probe is used to decrement the depth variable when a thread is dequeued.

```
solaris# ./rq.d
^C
[...]
       39
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  81527
               1 |@                                        1424
               2 |                                         21
               3 |                                         1
               4 |                                         1
               5 |                                         0

       47
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  81520
               1 |@                                        1523
               2 |                                         30
               3 |                                         0

       53
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  83952
               1 |@                                        1592
               2 |                                         28
               3 |                                         0
```

```
        44
            value  ------------- Distribution ------------- count
              < 0 |                                         0
                0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  84715
                1 |@                                        1607
                2 |                                         33
                3 |                                         0

        52
            value  ------------- Distribution ------------- count
              < 0 |                                         0
                0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  84833
                1 |@                                        1688
                2 |                                         35
                3 |                                         2
                4 |                                         0

        54
            value  ------------- Distribution ------------- count
              < 0 |                                         0
                0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  82882
                1 |@                                        1723
                2 |                                         37
                3 |                                         0
```

The sample output shows a very good balance of runnable threads on the run queues of each CPU. Each CPU typically has one to three runnable threads on its run queue during the sampling period.

CPUs and Interrupts

Observing interrupt activity can be important in understanding and maximizing performance. Modern I/O devices, such as 10Gb NIC cards and multiport 2/4/8Gb Fibre Channel HBAs, SATA controllers, and so on, can sustain very high rates of I/O, and interrupts are part of system I/O processing. Significant work went into the Solaris kernel to distribute interrupt load from fast devices to multiple CPUs and provide good out-of-the-box performance, but in some cases interrupt load can intrude on sustainable application throughput when application threads share a CPU that is handling a high rate of interrupts. `intrstat(1M)` is a DTrace consumer (command) included in Solaris which provides more detail on per-CPU interrupt load:

```
solaris# intrstat
      device |      cpu4 %tim      cpu5 %tim      cpu6 %tim      cpu7 %tim
-------------+------------------------------------------------------------
    e1000g#1 |         0  0.0     13252 51.1         0  0.0         0  0.0
      ehci#0 |         0  0.0         0  0.0         0  0.0         0  0.0

      device |      cpu4 %tim      cpu5 %tim      cpu6 %tim      cpu7 %tim
-------------+------------------------------------------------------------
    e1000g#1 |         0  0.0     12886 51.9         0  0.0         0  0.0
```

continues

```
      ehci#0 |          0   0.0          0   0.0          0   0.0          0   0.0
     emlxs#0 |          0   0.0          0   0.0          0   0.0          0   0.0
       mpt#0 |          0   0.0          0   0.0          0   0.0          0   0.0

      device |       cpu4  %tim       cpu5  %tim       cpu6  %tim       cpu7  %tim
-------------+-----------------------------------------------------------
       ata#0 |          0   0.0          0   0.0          3   0.0          0   0.0
    e1000g#1 |          0   0.0      13036 51.4          0   0.0          0   0.0
      ehci#0 |          0   0.0          0   0.0          0   0.0          0   0.0
     emlxs#0 |          0   0.0          0   0.0          0   0.0          0   0.0
       mpt#0 |          0   0.0          0   0.0          0   0.0          0   0.0
      ohci#0 |          9   0.1          0   0.0          0   0.0          0   0.0
```

The `intrstat(1M)` sample shown previously (edited for space) shows CPU 5 handling a high rate of interrupts from an e1000g network interface and spending just more than 50 percent of available cycles processing those interrupts.

The DTrace profile provider can be used to get another view on where CPU 5 is spending its time:

```
solaris# dtrace -n 'profile-997hz /arg0 && curthread->t_pri != -1
&& cpu == 5/ { @[func(arg0)] = count(); }'
dtrace: description 'profile-997hz ' matched 1 probe
[...]
  unix`do_splx                                                       126
  genunix`dblk_lastfree                                              128
  unix`kcopy                                                         137
  genunix`hcksum_assoc                                               166
  e1000g`e1000g_recycle                                              168
  unix`bcopy                                                         185
  mac_ether`mac_ether_header_info                                    204
  ip`tcp_rput_data                                                   301
  genunix`ddi_dma_sync                                               375
  e1000g`e1000g_send                                                 392
  unix`mutex_enter                                                   468
  e1000g`e1000g_receive                                              725
```

The one-liner shown previously is very similar to prior examples of profiling the kernel. In this example, we added a third expression to the predicate, testing for CPU 5. `cpu` is a global, built-in DTrace variable, and using it in a predicate in this way enables profiling a specific CPU, or a group of CPUs, by adding additional expressions to the predicate. This is extremely useful on systems with large CPU counts when the goal is to profile a subset of the available CPUs. The data here shows that the top kernel functions are the `e1000g_receive()` and `e1000g_send()` functions, along with the generic kernel `mutex_enter()` function. We can surmise that the kernel mutex lock is likely an e1000g driver-specific lock. With DTrace, it is easy to validate this.

```
solaris# dtrace -n 'mutex_enter:* { @s[stack()] = count(); }'
[...]
              ip`ipcl_classify_v4+0xa2
              ip`ip_tcp_input+0x757
              ip`ip_input+0xa1e
              dls`i_dls_link_rx+0x2c7
              mac`mac_do_rx+0xb7
              mac`mac_rx+0x1f
              e1000g`e1000g_intr+0x135
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
           355485

              ip`ipcl_classify_v4+0x14b
              ip`ip_tcp_input+0x757
              ip`ip_input+0xa1e
              dls`i_dls_link_rx+0x2c7
              mac`mac_do_rx+0xb7
              mac`mac_rx+0x1f
              e1000g`e1000g_intr+0x135
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
           355611

              ip`squeue_drain+0x175
              ip`squeue_enter_chain+0x394
              ip`ip_input+0xbff
              dls`i_dls_link_rx+0x2c7
              mac`mac_do_rx+0xb7
              mac`mac_rx+0x1f
              e1000g`e1000g_intr+0x135
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
           369351
```

The kernel stack trace captured when `mutex_enter()` is called validates that indeed the mutex lock calls are coming up from the e1000g interrupt handler (`e1000g_intr`), which seems reasonable given the system profile showing virtually all the kernel cycles spent in the e1000g driver code. If the presence of the kernel mutex lock function in the profile raises concerns over possible kernel lock contention, then `lockstat(1M)`, which is another DTrace consumer, will provide kernel lock statistics.

```
solaris# lockstat sleep 10

Adaptive mutex spin: 64453 events in 10.050 seconds (6413 events/sec)

Count indv cuml rcnt     nsec Lock                   Caller
-------------------------------------------------------------------------------
13393  21%  21% 0.00     1441 0xffffff112f01e428     e1000g_rxfree_func+0xaa
11494  18%  39% 0.00     1622 0xffffff112f01e428     e1000g_receive+0x337
  400   1%  39% 0.00     1157 0xffffff112f01e4e0     e1000g_send+0x1422
  395   1%  40% 0.00     1620 0xffffff117ad98810     putnext+0x70
```

continues

```
  361   1%  40% 0.00     1630 0xffffff117ae0d558     putnext+0x70
. . .
Adaptive mutex block: 575 events in 10.050 seconds (57 events/sec)

Count indv cuml rcnt     nsec Lock                   Caller
-------------------------------------------------------------------------------
  128  22%  22% 0.00    14830 0xffffff112f01e428     e1000g_receive+0x337
   40   7%  29% 0.00    34121 0xffffff112f01e428     e1000g_rxfree_func+0xaa
   29   5%  34% 0.00    11250 0xffffff112e8e1ac0     squeue_enter_chain+0x44
[...]
```

The `lockstat` data indicates that the top mutex spin events and mutex block events occurred in e1000g driver routines. The mutex spin data on the first line indicates a total of 19.3 milliseconds (13393 spins $\times$ 1441 nanoseconds per spin) was spent spinning on a mutex during the sampling period of 10 seconds.

CPU Events

Various system utilities, notably `mpstat(1M)` in Solaris, provide statistics on key metrics for each CPU. When observing CPUs, it is often useful to further track the sources of these statistics. The sysinfo provider (Solaris only) is a good place to start when using DTrace to track captured events.

```
solaris#  dtrace -qn 'sysinfo::: { @[probename] = count(); }
tick-1sec { printa(@); trunc(@); }'

[...]
  rawch                                                            1
  bwrite                                                           2
  lwrite                                                           3
  namei                                                           20
  outch                                                           20
  rw_rdfails                                                      46
  rw_wrfails                                                     117
  intrblk                                                        955
  trap                                                          1157
  lread                                                         2140
  inv_swtch                                                     3042
  sema                                                          4271
  mutex_adenters                                                5405
  idlethread                                                  116231
  xcalls                                                      158512
  readch                                                      176097
  sysread                                                     176098
  syswrite                                                    176882
  writech                                                     176882
  pswitch                                                     305054
```

The D program shown previously enables every probe managed by the sysinfo provider (execute `dtrace -l -P sysinfo` for a complete list). Using the tick provider, our simple command line provides per-second statistics. Here we see just more than 300,000 pswitch events per second (process switch, or context switches), followed by writech, syswrite, and sysread. We saw examples earlier of how to look deeper into context switch activity using the sched provider to determine which processes and threads are getting on-CPU. We can also determine why threads are being switched off-CPU by looking at user stacks and kernel stacks when the `sched:::off-cpu` probe fires.

```
solaris# dtrace -qn 'sched:::off-cpu /execname != "sched"/
{ @[execname, ustack()] = count(); } END { trunc(@,5); }'
^C
[...]
  oracle.orig
              libc.so.1`_read+0x8
              oracle.orig`nttfprd+0xac
              oracle.orig`nsbasic_brc+0x108
              oracle.orig`nioqrc+0x1a0
              oracle.orig`opikndf2+0x2b8
              oracle.orig`opitsk+0x2ec
              oracle.orig`opiino+0x3e8
              oracle.orig`opiodr+0x590
              oracle.orig`opidrv+0x448
              oracle.orig`sou2o+0x5c
              oracle.orig`opimai_real+0x130
              oracle.orig`ssthrdmain+0xf0
              oracle.orig`main+0x134
              oracle.orig`_start+0x17c
            22644
  oracle.orig
              libc.so.1`_read+0x8
              oracle.orig`nttfprd+0xac
              oracle.orig`nsbasic_brc+0x108
              oracle.orig`nioqrc+0x1a0
              oracle.orig`opikndf2+0x2b8
              oracle.orig`opitsk+0x2ec
              oracle.orig`opiino+0x3e8
              oracle.orig`opiodr+0x590
              oracle.orig`opidrv+0x448
              oracle.orig`sou2o+0x5c
              oracle.orig`opimai_real+0x130
              oracle.orig`ssthrdmain+0xf0
              oracle.orig`main+0x134
              oracle.orig`_start+0x17c
            22683
  oracle.orig
              libc.so.1`_pwrite+0x8
              libaio.so.1`_aio_do_request+0x1b4
              libc.so.1`_lwp_start
            27129
solaris# dtrace -qn 'sched:::off-cpu /execname != "sched"/
{ @[execname, stack()] = count(); } END { trunc(@,5); }'
^C
[...]
  oracle.orig
              unix`resume+0x4
              genunix`sema_p+0x138
              genunix`biowait+0x6c
              ufs`directio_wait_one+0x8
              ufs`directio_wait+0x34
              ufs`ufs_directio_write+0x900
```

continues

```
              ufs`ufs_write+0x174
              genunix`fop_write+0x20
              genunix`pwrite+0x22c
              unix`syscall_trap+0xac
             8973
    oracle.orig
              unix`resume+0x4
              genunix`cv_timedwait_sig_hires+0x190
              genunix`cv_waituntil_sig+0xb0
              semsys`semop+0x564
              unix`syscall_trap+0xac
            43064
    oracle.orig
              unix`resume+0x4
              genunix`cv_wait_sig+0x114
              genunix`str_cv_wait+0x28
              genunix`strwaitq+0x238
              genunix`kstrgetmsg+0xdcc
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
              genunix`read+0x274
              unix`syscall_trap+0xac
           999643
    rwdoit
              unix`resume+0x4
              genunix`cv_wait_sig+0x114
              genunix`str_cv_wait+0x28
              genunix`strwaitq+0x238
              genunix`kstrgetmsg+0xdcc
              sockfs`sotpi_recvmsg+0x2ac
              sockfs`socktpi_read+0x44
              genunix`fop_read+0x20
              genunix`read+0x274
              unix`syscall_trap+0xac
           999690
```

In the previous example, we have two very similar D programs executed from the command line. Our goal here again is to understand the pswitch metric (context switches) and why processes are getting switched off-CPU. In both programs, we use the `sched:::off-cpu` probe, which fires when a thread is switched off-CPU. The aggregation key in the first program tracks the user stack, and we can see the user call flow leading up to read and write systems calls. The second program changes `ustack()` as an aggregation key to `stack()` to go from a user view to a kernel view. The kernel view aligns with what we see in the user stack: Threads are being switched off because of blocking on read and write system calls. In the kernel stack sample, we also see instances of blocking on semaphore operations, and in the top kernel stack, we see blocking on a `pwrite(2)` system call to a UFS file. The bottom two kernel stacks indicate blocking on reads of a network socket.

The reads and writes observed using the sysinfo provider can be better understood using the syscall provider:

```
solaris# dtrace -qn 'syscall::*read:entry,syscall::*write:entry
{ @[probefunc] = count(); } tick-1sec { printa(@); trunc(@); }'

  pread                                                          1
  pwrite                                                       946
  read                                                      228018
  write                                                     228043

  pwrite                                                       962
  read                                                      225105
  write                                                     225142
```

This is an example of using a previous example to better understand the reads and writes:

```
solaris# dtrace -qn 'syscall::read:entry,syscall::write:entry
{ @[probefunc, fds[arg0].fi_fs] = count(); } tick-1sec { printa(@); trunc(@); }'

  read        proc                                                           16
  write       ufs                                                            30
  read        sockfs                                                     225418
  write       sockfs                                                     225418
```

In this example, we have two keys to the count aggregation: `probefunc`, which for this provider will be the name of the system call, and the `fds[].fi_fs` variable, showing the file system type of the file being read or written. The vast majority of the reads and writes are network (`sockfs`), with a much smaller number of writes going to a UFS file system, and reads from procfs (`/proc`).

Another event of interest observed from the sysinfo data is *xcalls*, or *cross calls*, which are CPU-to-CPU interrupts in Solaris. We can also observe per-CPU xcalls using `mpstat(1M)` and monitoring the `xcal` column:

```
solaris# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 2746  1707   48 4759  100 2464  130    7 17181   49  27   0  24
  1    0   0 2968  1808   49 5053   84 2560  115    8 18426   50  24   0  26
  2    0   0 2911  1742   50 5040   93 2610  116    7 18115   51  24   0  25
  3    0   0 2861  1729   41 4986   89 2498  110    6 18171   51  23   0  26
  4    0   0 2821  1786   50 5010   99 2527  117    7 17708   51  24   0  25
  5    0   0 2982  1958  276 4782   73 2443  104    5 17245   50  25   0  25
[...]
```

In Solaris, cross calls are relatively lightweight events—modern CPUs are capable of handling large volumes of xcalls per second. The per-CPU statistics we see from `mpstat(1M)` show about 3,000 xcalls per second per CPU, which is not a large number for a modern, busy system. The source of the xcalls can be determined by examining the kernel stack when the `xcalls` probe of the sysinfo provider fires:

```
solaris# dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); }'
[...]
              FJSV,SPARC64-VII`send_one_mondo+0x20
              unix`xt_one_unchecked+0xc8
              genunix`sleepq_wakeall_chan+0x48
              genunix`cv_broadcast+0x4c
              ip`tcp_fuse_output+0x7f0
              ip`tcp_output+0x74
              ip`squeue_drain+0x130
              ip`squeue_enter+0x348
              sockfs`sostream_direct+0x194
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
           177428

              FJSV,SPARC64-VII`send_one_mondo+0x20
              unix`xt_one_unchecked+0xc8
              genunix`sleepq_wakeall_chan+0x48
              genunix`cv_broadcast+0x4c
              ip`tcp_fuse_output+0x7f0
              ip`tcp_output+0x74
              ip`squeue_enter+0x74
              sockfs`sostream_direct+0x194
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
          1177168
```

Examining the kernel stack frame, starting at the bottom, we can see that most xcalls originate from network writes, initiating a wake-up to sleeping threads.

The same method can be applied to chasing any of the other sysinfo events observed. Here is another example:

```
solaris# dtrace -n 'sysinfo:::sema { @[stack()] = count(); }'
dtrace: description 'sysinfo:::sema ' matched 1 probe
^C


              semsys`semop+0x28
              unix`syscall_trap+0xac
            25508

solaris# dtrace -n 'sysinfo:::sema { @[execname] = count(); }'
dtrace: description 'sysinfo:::sema ' matched 1 probe
^C

  oracle.orig                                                    23453
```

In the previous example, there are two `dtrace(1M)` invocations. The first is essentially the same as the xcalls example, only this time the probe name is changed to `sema` (semaphore operation), corresponding to the specific sysinfo event of interest. The second is the same probe, only this time the aggregation key is the DTrace `execname` variable, rather than the `stack()`. The results are relatively simple; the kernel stack shows the sema events are the result of semop system calls, and the semop system calls are being generated by the `oracle.orig` processes. By changing the aggregation key to use `pid` instead of `execname`, we can determine whether a particular process is making an inordinate number of semop system calls and drill down further by obtaining a user stack frame when a semop system call is executed by a specific process.

```
solaris# dtrace -n 'sysinfo:::sema { @[pid] = count(); }'
dtrace: description 'sysinfo:::sema ' matched 1 probe
^C
[...]
    27924              240
    27760              241
    27926              241
    28006              241
    28028              242
     2674             1954

solaris# dtrace -n 'syscall::semsys:entry { @[pid] = count(); }'
dtrace: description 'syscall::semsys:entry ' matched 1 probe
^C
[...]
    28006              244
    27858              245
    27760              247
    28028              248
     2674             2028

solaris# dtrace -n 'syscall::semsys:entry /pid == 2674/ { @[ustack()] = count(); }'
dtrace: description 'syscall::semsys:entry ' matched 1 probe
^C
[...]
              libc.so.1`_syscall6+0x1c
              a.out`_$c1A.kslpstevent+0x7fc
              oracle.orig`kcrfw_post+0x95c
              oracle.orig`kcrfw_redo_write+0xd34
              oracle.orig`ksbabs+0x58c
              oracle.orig`ksbrdp+0x4cc
              oracle.orig`opirip+0x454
              oracle.orig`opidrv+0x308
              oracle.orig`sou2o+0x5c
              a.out`opimai_real+0x204
              oracle.orig`ssthrdmain+0xf0
              a.out`main+0x134
              a.out`_start+0x17c
             3695
```

Here again we have three consecutive DTrace executions to illustrate the potential for rapid drill-down. First we use the sema probename in the sysinfo provider and use `pid` as an aggregation key. We can see process PID 2674 had the largest number of semaphore calls during the sampling period. Next we use a different probe—the entry point to the semaphore system call, taking the same action. This illustrates how, in some cases, we can use more than one probe to gather the same basic information. Finally, in the third example, we add a predicate to take action only when PID 2674 executes a semaphore system call, and we key on the user stack so we can see the user code path leading to the semaphore call.

We can apply the same method for understanding the source of CPU statistics and events by grouping related events together with multiple probe specifications.

```
solaris# dtrace -qn 'sysinfo:::sysread,sysinfo:::readch
{ @[pid, probename] = count(); } END { trunc(@, 20); exit(0); }'
^C
    15533  readch                                                          38033
    15533  sysread                                                         38033
    15575  readch                                                          38033
    15575  sysread                                                         38033
    15554  readch                                                          39370
    15554  sysread                                                         39370
    15569  readch                                                          39370
    15569  sysread                                                         39370
    15434  readch                                                          39393
    15434  sysread                                                         39393
    15481  readch                                                          39394
    15481  sysread                                                         39394
    15604  readch                                                          39744
    15604  sysread                                                         39744
    15653  readch                                                          39744
    15653  sysread                                                         39744
    15451  readch                                                          39811
    15479  readch                                                          39811
    15479  sysread                                                         39811
    15451  sysread                                                         39812
```

As always, the columns are ordered based on the aggregation key ordering, so starting at the left, we see the process PID that was on-CPU when the probe fired, the probe name, and the count value. This information shows us which processes issued the most read calls during the tracing period.

CPU Summary

The metrics of interest when observing CPUs are utilization and saturation—to see whether the CPUs are a contended resource. DTrace can determine which workload processes and threads are using CPU cycles, down to the software functions responsible—in both user-land and the kernel. As is the case with all things performance, latency is a critical measurement, and CPU latency (both time waiting for a CPU and time running on a CPU) is measurable with DTrace, as shown in this section.

Observing Memory

Several utilities are available for monitoring memory systemwide, as well as per-process virtual and physical memory. Which you should use depends on the memory problem or capacity issue observed or whether your goal is simply to account for memory use.

With DTrace, we can do the following:

- Dynamically monitor kernel memory allocation and determine which kernel subsystem is consuming memory
- Dynamically monitor process memory allocation and determine where in user code memory allocation is originating
- Correlate system-reported memory paging activity to the application processes generating the observed events (page faults, page-ins, page-outs)
- Measure the memory-related latency in terms of how much time application processes are waiting for memory allocations, page faults, and other memory events

Depending on what the problem is and what's been observed, DTrace can be used to drill down on all aspects for memory allocation and use.

Memory Strategy

When examining memory use, consider the main consumers of physical memory.

- **The kernel**: This includes executable code, I/O buffers, and system metadata.
- **File system caches**: These are technically part of the kernel but potentially a large enough consumer to be treated separately. The file system being used (UFS, ZFS) will determine the right method for measuring this.
- **User processes**: Application code and heap allocations.

The first step in observing memory is to get a big-picture view of how much memory is being used by each of these consumers.

Begin by checking whether a systemwide memory deficit is occurring, which should be possible via the operating system `vmstat(1M/8)` tool (or `vm_stat(1)` on Mac OS X). This may be identified by checking how much free memory the system reports as available, if the page scanner is running, and if unwanted page-in/page-out activity is occurring. System performance degrades substantially when the kernel needs to continually locate pages for freeing and move active memory pages on and off the physical swap devices.

Once systemwide memory usage is understood, further investigation can be done, looking at the various memory consumers and understanding requirements.

Memory Checklist

The checklist in Table 3-5 provides guidelines to observing memory.

Memory Providers

Table 3-6 lists the DTrace providers applicable to observing memory.

The vminfo, io, pid, and plockstat providers are not yet available on FreeBSD.

Table 3-5 Memory Checklist

Issue	Description
Systemwide memory shortfall	The system does not have sufficient physical memory to support the workload, resulting in page-in/page-out activity. All operating systems generally perform very poorly when memory becomes a contended resource and there is sustained paging activity.
Kernel memory allocation	Observed metrics indicate CPU cycles and memory consumption by the kernel.
User process memory allocation	Observed metrics indicate CPU cycles in user memory allocation/deallocation and memory consumption.
Memory page-in/page-out activity	Virtual memory (VM) statistics indicate memory page-in/page-out activity.
Memory pagefault activity	VM statistics indicate minor and/or major page fault activity. Major page faults require a disk I/O; minor pagefaults do not.

Table 3-6 Memory Providers

Provider	Description
vminfo	Virtual memory statistics
syscall	Processes memory allocation system calls (`brk(2)`, `mmap(2)`)
io	Observes disk I/O due to paging or swapping
fbt	Kernel functions related to memory allocation, deallocation, virtual memory, and page management

Table 3-6 Memory Providers (*Continued*)

Provider	Description
pid	User process memory usage (`trace malloc()`, and so on)
plockstat	User process locks related to memory routines

Memory One-Liners

The following one-liners can be used as a starting point for memory analysis. As is the case with any DTrace one-liner, they can be inserted into a file and turned into a DTrace script to facilitate adding probes, predicates, additional data to collect, and so on.

vminfo Provider

Tracking memory page faults by process name:

```
dtrace -n 'vminfo:::as_fault { @mem[execname] = sum(arg0); }'
```

Tracking pages paged in by process name:

```
dtrace -n 'vminfo:::pgpgin { @pg[execname] = sum(arg0); }'
```

Tracking pages paged out by process name:

```
dtrace -n 'vminfo:::pgpgout { @pg[execname] = sum(arg0); }'
```

sched Provider

Tracking process user stack sizes:

```
dtrace -n 'sched:::on-cpu { @[execname] = max(curthread->t_procp->p_stksize);}'
```

Tracking which processes are growing their address space heap segment:

```
dtrace -n 'fbt::brk:entry { @mem[execname] = count(); }'
```

fbt Provider

Tracking which processes are growing their address space stack segment:

```
dtrace -n 'fbt::grow:entry { @mem[execname] = count(); }'
```

pid Provider

These use the pid provider to trace a given process ID (PID). Either `-p PID` or `-c 'command'` can be used to specify the process.

Process allocation (via malloc()) counting requested size:

```
dtrace -n 'pid$target::malloc:entry { @[arg0] = count(); }' -p PID
```

Process allocation (via malloc()) requested size distribution plot:

```
dtrace -n 'pid$target::malloc:entry { @ = quantize(arg0); }' -p PID
```

Process allocation (via malloc()) by user stack trace and total requested size:

```
dtrace -n 'pid$target::malloc:entry { @[ustack()] = sum(arg0); }' -p PID
```

Process allocation (via `malloc()`) by Java stack trace and total requested size:

```
dtrace -n 'pid$target::malloc:entry { @[jstack()] = sum(arg0); }' -p PID
```

Memory Analysis

Memory analysis begins with getting a systemwide view of key memory metrics. Of course, it's useful to know how much physical memory is installed in the system and how much swap space is configured. On Solaris systems, these first basic steps can be accomplished using the following:

```
solaris_sparc# prtconf | grep "Memory size"
Memory size: 131072 Megabytes
solaris_sparc# swap -l
swapfile              dev  swaplo blocks   free
/dev/md/dsk/d20     85,20     16 20484272 20474544
/dev/dsk/c1t0d0s1   32,9      16 143074544 143061744
```

```
solaris_x86# prtconf | grep "Memory size"
Memory size: 10231 Megabytes
solaris_x86# swap -l
swapfile             dev  swaplo blocks   free
/dev/dsk/c0t0d0s1   32,1       8 1060280 1060280
```

There are two examples shown previously—the first from a Solaris SPARC system and the second from a Solaris x86 system. Gathering this information as a starting point may seem rudimentary, but it's important to know basic system information before proceeding with any analysis.

The next step is to get a systemwide view of memory usage and how much memory is free. In Solaris, this is most easily accomplished using the `memstat` dcmd (d-command) under `mdb(1)`:

```
solaris# echo "::memstat" | mdb -k
Page Summary                Pages                MB  %Tot
------------     ----------------  ----------------  ----
Kernel                     268971              2101    2%
Anon                      4761502             37199   29%
Exec and libs               30866               241    0%
Page cache                3576963             27945   22%
Free (cachelist)          2010595             15707   12%
Free (freelist)           5839192             45618   35%

Total                    16488089            128813
Physical                 16466389            128643
```

In Solaris 10 5/09 or earlier, running `memstat` on a large system can take many minutes. In Solaris 10 10/09, `memstat` was enhanced and made much faster.

Other utilities provide a good starting metric for looking at memory. `vmstat(1M/8)` in both Solaris and FreeBSD allows free memory to be observed and provides an indication of memory shortfalls with the `sr` (scan rate) column. In Mac OS X, `vm_stat(1)` offers a similar systemwide view based on its own virtual memory implementation.

```
solaris# vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr m1 m1 m1 m2   in   sy   cs us sy id
 0 0 0 148132968 72293376 3 11 2 0 0  0  0 14 14 14  1 13671 30379 23848 2 1 96
 0 2 0 138621040 59492152 9 97 0 0 0  0  0 161 160 161 0 243611 503049 436315 46 25 28
 4 1 0 138620616 59491792 0 30 0 0 0  0  0 189 189 190 0 241980 508303 434517 47 25 27
 4 1 0 138620616 59491776 0 0 0 0  0  0  0 142 142 141 0 239497 507814 431138 46 25 29
[...]
```

The previous Solaris `vmstat(1M)` sample shows a system with a substantial amount of free memory (59GB). The `sr` column (page scan rate) is zero. Nonzero sr column values indicate the kernel had to nudge the page scanner to wake up and start finding memory pages to steal to replenish the memory freelist.

```
macosx# vm_stat 1
Mach Virtual Memory Statistics: (page size of 4096 bytes, cache hits 0%)
  free active   spec inactive   wire  faults     copy    0fill reactive  pageins pageout
 31010 550907 19312   205106 209483 158595K 2278296 52701106   211197  1305059 51672
 31026 550933 19312   205106 209483      53        1       35        0        0 0
 31040 550937 19312   205106 209483      26        0       12        0        0 0
 31057 550934 19312   205106 209483      23        0        8        0        0 0
 30820 551164 19312   205106 209483     257        0      243        0        0 0
 30851 551152 19312   205106 209483      27        0       13        0        0 0
```

The values displayed by vm_stat(1) in Mac OS X are pages (showing the page size in the output header), providing a systemwide view of key memory statistics.

Per-process memory use can be monitored on Solaris systems using either the prstat(1) or ps(1) command. prstat(1) lets you sort the output based on resident memory size for each process, making it easier to see which processes are the largest memory consumers.

```
solaris# prstat -s rss -n 20 -c 1
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  1818 noaccess  149M  119M sleep   59    0   0:00:49 0.0% java/18
     7 root       13M   10M sleep   59    0   0:00:02 0.0% svc.startd/12
     9 root       11M 9944K sleep   59    0   0:00:09 0.0% svc.configd/16
   615 root       13M 9180K sleep   59    0   0:00:33 0.0% fmd/21
   156 root     9696K 6636K sleep   59    0   0:00:03 0.0% nscd/33
  2984 mauroj   6788K 4152K sleep   59    0   0:00:00 0.0% sshd/1
   606 root     4724K 3060K sleep   59    0   0:00:00 0.0% inetd/4
  2998 root     3724K 2784K cpu2    59    0   0:00:00 0.0% prstat/1
   488 root     3852K 2664K sleep   59    0   0:00:00 0.0% automountd/2
  2983 root     4808K 2648K sleep   59    0   0:00:00 0.0% sshd/1
   150 root     3660K 2364K sleep   59    0   0:00:00 0.0% picld/4
   124 root     5460K 2076K sleep   59    0   0:00:00 0.0% syseventd/15
   143 daemon   4228K 2036K sleep   59    0   0:00:00 0.0% kcfd/3
   732 root     3008K 2032K sleep   59    0   0:00:00 0.0% vold/4
   875 root     9012K 1992K sleep   59    0   0:00:01 0.0% sendmail/1
   147 root     3596K 1876K sleep   59    0   0:00:00 0.0% devfsadm/7
   546 root     3868K 1824K sleep   59    0   0:00:00 0.0% syslogd/11
  2995 root     2900K 1780K sleep   59    0   0:00:00 0.0% bash/1
   814 smmsp    9076K 1732K sleep   59    0   0:00:00 0.0% sendmail/1
  2214 root     5592K 1676K sleep   59    0   0:00:00 0.0% dtlogin/1
Total: 45 processes, 194 lwps, load averages: 0.00, 0.00, 0.00
```

The RSS column shows the physical memory usage of the process (SIZE is the virtual memory size). Be aware of workloads that make use of shared memory, because each process will show physical memory usage that includes shared memory. Here's an example:

```
solaris# prstat -s rss -c 1
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
  2662 oracle     38G   38G sleep   31    0  19:03:55 0.1% oracle.orig/1
  2668 oracle     38G   38G sleep   51    0   6:11:27 0.0% oracle.orig/258
  2666 oracle     38G   38G sleep   41    0   6:10:52 0.0% oracle.orig/258
  2670 oracle     38G   38G sleep   50    0   6:11:58 0.0% oracle.orig/258
  2672 oracle     38G   38G sleep   43    0   6:16:39 0.0% oracle.orig/258
  2682 oracle     38G   38G sleep   51    0   0:08:59 0.0% oracle.orig/11
  2678 oracle     38G   38G sleep   59    0   0:09:16 0.0% oracle.orig/39
  2676 oracle     38G   38G sleep   41    0   1:21:03 0.0% oracle.orig/19
  2714 oracle     38G   38G sleep   59    0   0:10:08 0.0% oracle.orig/1
  2648 oracle     38G   38G sleep   52    0   0:49:03 0.0% oracle.orig/1
  2700 oracle     38G   38G sleep   59    0   0:01:48 0.0% oracle.orig/1
[...]
```

As is typical with database systems, all the processes associated with the database instance attach to the same shared memory segment, so each process shows a physical memory size of 38GB. If we were to sum the RSS column for all processes, the resulting value would far exceed the amount of physical memory installed, because most of that 38GB is a shared memory segment that is part of each process's address space. Only one copy of those physical memory pages is actually resident in memory. Note that this applies to other types of shared memory, such as shared libraries.

User Process Memory Activity

The commands shown previously provide the big picture for determining physical memory allocation and use by user processes. When analyzing memory, in addition to observing the actual amount of physical memory being used by processes, it's important to track which processes are allocating and freeing memory and whether processes are waiting for memory allocations or waiting for their memory pages to be paged in.

The vast majority of memory consumption by user processes is for heap space. Typically, when examining the physical memory usage of a user process, the heap segment will dominate. Space for stack segments, especially for processes with a large number of threads, may be large, and of course the size of the text segment will vary based on the size of the executable. User process heap segments are allocated based on API calls to the `malloc(3)` family of interfaces or using `mmap(2)`. The underlying implementation will vary greatly across different operating systems. Solaris, for example, typically uses the `brk(2)` system call underneath `malloc(3)` calls when physical memory allocation is needed. The Solaris `grow()` kernel function is called for growing other address space segments.

Mac OS X does not appear to implement the `brk(2)` system call, so the underlying mechanism used to allocate heap segments is not readily apparent. Let's use DTrace to see whether we can figure out how Mac OS X implements `malloc(3)` by examining a known workload. The workload was a simple program written in C that makes eight `malloc(3)` calls in a loop, starting with malloc'ing 100MB and adding 100KB to each subsequent `malloc(3)`. We'll use the DTrace syscall provider to see which underlying system calls are used:

```
macosx# dtrace -n 'syscall:::entry /pid == $target/
{ @[probefunc]=count(); }' -c ./mm
malloc of 104857600 done, touching pages...
dtrace: description 'syscall:::entry ' matched 434 probes
malloc of 104960000 done, touching pages...
malloc of 105062400 done, touching pages...
malloc of 105164800 done, touching pages...
malloc of 105267200 done, touching pages...
malloc of 105369600 done, touching pages...
malloc of 105472000 done, touching pages...
malloc of 105574400 done, touching pages...
dtrace: pid 6283 has exited

  exit                                                              1
  mmap                                                              7
  write_nocancel                                                    7
  madvise                                                          16
```

The command line shown previously enables a probe on the entry point of all system calls when the PID of the `mm` process is the PID on-CPU when the probe fires (mm is our test program). The `malloc` messages are generated by the program (not by DTrace). The result shows 16 calls to `madvise(2)` and 7 calls to `mmap(2)` as memory-related system calls. Referencing the man pages, we know `madvise(2)` is not used to allocate memory, but `mmap(2)` most certainly is, so it seems `mmap(2)` is used in Mac OS X to enter the kernel for heap allocation. Let's verify this by just tracing `mmap(2)` calls and tracking the allocation size:

```
macosx# dtrace -n 'syscall::mmap:entry /pid == $target/
{ @[arg1] = count(); }' -c ./mm
malloc of 104857600 done, touching pages...
dtrace: description 'syscall::mmap:entry ' matched 1 probe
malloc of 104960000 done, touching pages...
malloc of 105062400 done, touching pages...
malloc of 105164800 done, touching pages...
malloc of 105267200 done, touching pages...
malloc of 105369600 done, touching pages...
malloc of 105472000 done, touching pages...
malloc of 105574400 done, touching pages...
dtrace: pid 6292 has exited

        104960000              1
        105062400              1
```

```
        105164800              1
        105267200              1
        105369600              1
        105472000              1
        105574400              1
```

We changed the DTrace script to just enable a probe at the `mmap(2)` entry point, and we tracked the second argument passed to `mmap(2)` in our action (reference the `mmap(2)` man page—the second argument is the size). We can see that size arguments passed to `mmap(2)` align precisely with what our test program passes to `malloc(3)`, so we can conclude that Mac OS X creates user process heap segments using `mmap(2)`, most likely using the `MAP_ANON` flag to instruct `mmap(2)` to map an anonymous memory segment (heap) vs. mapping a file. Of course, we can use DTrace to verify this:

```
macosx# dtrace -qn 'syscall::mmap:entry /pid == $target/
{ printf("FLAG: %x\n", arg3); }' -c ./mm
malloc of 104857600 done, touching pages...
[...]
FLAG: 1002
FLAG: 1002
[...]
```

We changed the DTrace program to use a `printf` statement in the action and print the value of `arg3` in hexadecimal. We know from `man mmap(2)` that the fourth argument is the flags, and the flags are defined in the `/usr/include/sys/mman.h` header file as hex values, so printing them in hex makes it easier to find our answer:

```
macosx# grep MAP_ANON /usr/include/sys/mman.h
#define     MAP_ANON    0x1000      /* allocated from memory, swap space */
```

The DTrace output shows 0x1002 as the flag argument to `mmap(2)`, and an examination of the header file shows that, indeed, 0x1000 means the `MAP_ANON` flag is set (the 0x0002 flag is left as an exercise for the reader).

If we want to further drill down with DTrace and determine which OS X kernel function is called for memory allocation, we can use the fbt provider and the following script:

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent
```

continues

```
syscall::mmap:entry
{
self->flag = 1;
}
fbt:::
/self->flag/
{
}
syscall::mmap:return
/self->flag/
{
   self->flag = 0;
   exit(0);
}
```

Script mmap.d

The `mmap.d` script enables all the fbt provider probes. When the `mmap(2)` system call is entered, we set a flag that is used as a predicate in the fbt probes. Since an action is not specified, we'll get the default output of the CPU and probe function when the fbt probes fire. Using the `flowindent` DTrace option, we'll generate an easy-to-read kernel function call flow.

```
macosx# ./mmap.d -c ./mm
malloc of 104857600 done, touching pages...
dtrace: script './mmap.d' matched 18393 probes
malloc of 104960000 done, touching pages...
CPU FUNCTION
  1  -> mmap
  1    -> current_map
  1    <- current_map
  1    -> vm_map_enter_mem_object
  1      -> vm_map_enter
  1        -> lock_write
  1        <- lock_write
  1        -> vm_map_entry_insert
  1          -> zalloc
  1            -> zalloc_canblock
  1              -> lck_mtx_lock_spin
  1              <- lck_mtx_lock_spin
  1              -> lck_mtx_unlock_darwin10
  1              <- lck_mtx_unlock_darwin10
  1            <- zalloc_canblock
  1          <- zalloc
  1        <- vm_map_entry_insert
  1        -> lock_done
  1        <- lock_done
  1        -> lck_rw_done_gen
  1        <- lck_rw_done_gen
  1      <- vm_map_enter
  1    <- vm_map_enter_mem_object
  1  <- mmap
  1  <= mmap
```

The call flow generated enables us to observe key kernel functions called for memory allocations, `vm_map_enter()` and `zalloc()`. The `zalloc()` function sounds interesting (allocate zeroed memory pages is our guess) and can be instrumented to track memory allocations by processes. However, since we're not certain under what other circumstances the Mac OS X `zalloc()` kernel function may be called, we can stick with using a DTrace probe on `mmap(2)`.

```
macosx# dtrace -qn 'syscall::mmap:entry /arg3 & 0x1002/
{ @m[execname, arg1] = count(); }'
^C

  Terminal                                                    4096                1
  dtrace                                                      4096                1
  dtrace                                                    266240                1
  dtrace                                                   4194304                2
  Dock                                                       23040               21
  Dock                                                       90112               21
  WindowServer                                               16384               21
```

The DTrace program used here tracks the process name and size of processes calling `mmap(2)`. We use a predicate to determine whether the `MAP_ANON` flag is set, since our interest here is tracking `mmap(2)` calls for heap memory allocations vs. use of `mmap(2)` for mapping regular files. We also test for 0x0002, which is `MAP_PRIVATE`, further reducing our output to only those allocations for private, anonymous memory segments. During the sample period, we can see the Mac OS X `WindowServer` process did 21 `mmap(2)` calls for 16KB chunks of memory.

In Solaris, the `brk(2)` system call is typically invoked when a user code calls `malloc(3)` (which is a user library function), so we can begin there to see which processes are doing memory allocations. It is important to note here that there are many memory allocators available for Solaris, implemented as binary-compatible `malloc(3)` calls in different shared object libraries. `libc.so`, `libmtmalloc.so`, `libumem.so`, and so on, can be linked to your Solaris code to make use of a different implementation of `malloc(3)` that may be more suitable in terms of performance and/or efficiency. The `libc.so` `malloc(3)` uses `brk(2)`—`malloc(3)` out of other shared object libraries may use a different underlying mechanism to enter the kernel for memory allocation.

```
solaris# dtrace -n 'syscall::brk:entry { @[pid,execname] = count(); }'
^C
[...]
    21008  arch                                                          28
    21013  arch                                                          28
    20958  oracle                                                        48
    21034  m2loader                                                     125
```

continues

```
    20827  java                                                                  246
    21035  java                                                                  332
    21033  java                                                                  340
```

The output shows several Java processes doing memory allocations. The `brk(2)` system call in Solaris does not take a size argument, so we cannot easily determine the amount of memory requested in Solaris by using a probe on `brk(2)`. However, it is possible to track heap growth by doing some math on the address passed to `brk(2)`, tracked over multiple calls.

```
1  #!/usr/sbin/dtrace -qs
2
3  self int endds;
4
5  syscall::brk:entry
6  /pid == $target && !self->endds/
7  {
8       self->endds = arg0;
9  }
10
11 syscall::brk:entry
12 /pid == $target && self->endds != arg0/
13 {
14      printf("Allocated %d\n", arg0 - self->endds);
15      self->endds = arg0;
16 }
```

Script brk.d

The `brk(2)` system call takes a memory address as the only argument (`arg0`). The predicate for the first action clause (line 6) verifies the PID specified on the command line and tests that the thread-local variable `self->endds` is zero, in which case `self->endss` is set to `arg0`, which is the address passed to `brk(2)`. In the second predicate (line 12), if the passed address (`arg0`) is different from the previous address, we've been through at least one pass of `brk(2)`, so we can measure the address space growth by finding the difference between the previous and current memory address (line 14). Here's a sample run on a test program that malloc's 8KB memory segments:

```
solaris# ./brk.d -c ./a.out
Allocated 16384
Allocated 8192
Allocated 8192
Allocated 8192
Allocated 8192
Allocated 8192
^C
```

Another approach is to track sizes from a different level in the software stack. Specifically, we can instrument the `malloc(3)` library call in processes of our choice, but doing so requires the use of the pid provider since `malloc(3)` is a library interface and resides in user space, not in the kernel.

The system we're looking at for this example is running a workload that creates/terminates Java processes (Java virtual machines [JVMs]) fairly rapidly, so using the pid provider and specifying a PID is tricky. We found that most of the time, the Java process we targeted had exited by the time we start the DTrace. To complete the example, we used `pgrep(1)` to grab the most recent running Java process.[4]

```
solaris# dtrace -n 'pid$target::malloc:entry
{ @[pid, arg0] = count(); }' -p `pgrep -nx java`
dtrace: description 'pid$target::malloc:entry ' matched 2 probes
dtrace: pid 11089 has exited

    11089              552                1
    11089             1480                1
    11089              256                2
```

The `java` process we tracked here (PID 11089) did just a few `malloc(3)` calls for relatively small amounts of memory. We could take this one step further and examine the call stack leading to the malloc calls.

```
solaris# dtrace -n 'pid$target::malloc:entry
{ @[jstack()] = count(); }' -p `pgrep java | tail -1`
dtrace: description 'pid$target::malloc:entry ' matched 2 probes
^C


              libc.so.1`lmalloc
              libc.so.1`fdopendir+0x22
              libc.so.1`opendir+0x3e
              libjava.so`Java_java_io_UnixFileSystem_list+0x65
              0xf82bfcf8
              0xeff0dc18
              0xebefeaf8
                6

              libc.so.1`lmalloc
              libc.so.1`fdopendir+0x30
              libc.so.1`opendir+0x3e
              libjava.so`Java_java_io_UnixFileSystem_list+0x65
              0xf82bfcf8
              0xeff0dc18
              0xebefeaf8
                6
```

4. We hasten to point out that the DTrace will generate output only if `java` processes are executing. Change the argument to `-p` accordingly to suit your needs.

Note the use of the `jstack()` function in the DTrace program executed previously. Note also that the stack frames have been partially resolved. That is, some entries show hex numbers (user virtual memory addresses) and not symbol names. DTrace collects and stores the stack frames in their raw, numeric form and converts addresses to symbols only when it's time to display output. In some cases, DTrace may not be able to convert a user address to its corresponding symbol. This can happen if the process exits during tracing or the symbolic information has been stripped.

In the previous example, we can see that the traced JVM was entering malloc from a file system I/O code path. This gives us a little more insight into what the code is doing, should it be necessary to drill down further.

Here's another user process example, this time monitoring malloc calls from Firefox on Mac OS X:

```
macosx> dtrace -n 'pid$target::malloc:entry
{ @[ustack()] = quantize(arg0); }' -p 1806
^C
[...]

              libSystem.B.dylib`malloc
              libmozjs.dylib`js_ValueToCharBuffer+0x3f55
              libmozjs.dylib`js_ValueToCharBuffer+0x4004
              libmozjs.dylib`js_ValueToCharBuffer+0x3b4
              libmozjs.dylib`js_CoerceArrayToCanvasImageData+0x183c
              libmozjs.dylib`js_CoerceArrayToCanvasImageData+0x1e54
              libmozjs.dylib`JS_HashTableRawRemove+0xab14
              libmozjs.dylib`js_FreeStack+0x18f8
              libmozjs.dylib`JS_EvaluateUCScriptForPrincipals+0x9b
             [...]
              XUL`DumpJSStack+0x17b5f2
              XUL`DumpJSStack+0x17f075

           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |@                                        8
             256 |@@                                       16
             512 |@@                                       16
            1024 |@@@                                      27
            2048 |@@@                                      33
            4096 |@@@@                                     39
            8192 |@@@@@@@@@@                               95
           16384 |@@@@@@@@@                                87
           32768 |@@@@@@                                   63
           65536 |@@                                       15
          131072 |                                         0

              libSystem.B.dylib`malloc
              XUL`cmmf_decode_process_cert_response+0x2cdc3
              XUL`cmmf_decode_process_cert_response+0x21f14
              XUL`cmmf_decode_process_cert_response+0x242ec
              XUL`cmmf_decode_process_cert_response+0x2447b
              XUL`cmmf_decode_process_cert_response+0x365
            [...]
              XUL`JSD_GetValueForObject+0xebab9
              XUL`JSD_GetValueForObject+0xe3602
```

```
              XUL`JSD_GetValueForObject+0xf5d33
              AppKit`-[NSView _drawRect:clip:]+0xdb6
              AppKit`-[NSView _recursiveDisplayAllDirtyWithLockFocus:visRect:]+0x640

           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@@@@                                  2
            4096 |@@@@@@@@@@@@@                            4
            8192 |@@@                                      1
           16384 |@@@                                      1
           32768 |                                         0
           65536 |@@@@@@@                                  2
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |@@@@@@@                                  2
         2097152 |                                         0
```

In the previous example, we determined the PID of Firefox on Mac OS X and tracked `malloc()` requested sizes using `quantize` along with aggregating on user stack traces. With this running, we did a couple of Web page loads with Firefox. This generated a huge amount of output, because Firefox does a great deal of memory allocation using `malloc()`. In the two sample frames, the bottom frame shows a small number of malloc calls (count values), ranging from 2KB to 2MB. The top frame shows a larger number of malloc calls, most in the 8KB to 64KB range.

Also of interest when monitoring memory is tracking kernel virtual memory events to the processes that are generating or waiting for those events. The DTrace vminfo provider makes it relatively simple to correlate virtual memory events to processes and quantify the effects on performance. First, get the big picture with `vmstat(1)` on Solaris:

```
solaris# vmstat 1
  kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s0 s1 s2 ---   in   sy   cs us sy id
 0 0 0 15377184 24309708 278 1546 243 0 0 0 0 15 15  0 0 4020 5314 2630 2 1 96
 3 0 0 4373156 12970972 2681 4728 9959 0 0 0 0 22 22 0 0 18460 4070 5312 5 5 90
 0 0 0 4365792 12969308 107 174 803 0 0 0 0 22 22 0  0 4029 3158 3691  1  3 96
 2 0 0 4365772 12969236 17 280 241 0 0 0 0 10 10  0  0 3883 4392 3999  7  3 90
 0 0 0 4320928 12968464 4025 23844 619 0 0 0 0 72 70 0 0 24525 20056 5212 27 16 57
 1 0 0 4428636 12935872 21 565 157 0 0 0 0 190 187 0 0 5377 187861 5265 16 7 77
 0 0 0 4426576 12933464 21 4238 167 0 0 0 0 5  5  0  0 3633 90565 2963 22 4 73
^C
```

Next, use the vminfo provider to correlate VM events and running processes:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

```

continues

```
5  vminfo:::
6  {
7          @[execname, probefunc, probename] = count();
8  }
9  tick-1sec
10 {
11         trunc(@);
12         printf("%-16s %-16s %-16s %-8s\n", "EXEC", "FUNCTION", "NAME", "COUNT");
13         printa("%-16s %-16s %-16s %-@8d\n",@);
14         trunc(@);
15         printf("\n");
16 }
```

Script vmtop.d

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  vminfo:::
6  {
7          @[execname, probefunc, probename] = count();
8  }
9  tick-1sec
10 {
11         trunc(@, 10);
12         printf("%-16s %-16s %-16s %-8s\n", "EXEC", "FUNCTION", "NAME", "COUNT");
13         printa("%-16s %-16s %-16s %-@8d\n",@);
14         trunc(@);
15         printf("\n");
16 }
```

Script vmtop10.d

The `vmtop10.d` script enables every probe managed by the vminfo provider and aggregates on `execname`, `probefunc`, and `probename` as keys. The script generates output every second (line 9). We use a `printf()` statement (line 12) to label the columns, and we manage column width and alignment in the `printf()` and `printa()` statements in order to produce output that is easier to read.

The `probefunc` (`FUNCTION` heading) shows us where the probe resides in the kernel and can be used if further analysis is required by using the fbt provider to instrument those functions or reading the source code. We also truncate the aggregation collected, using the DTrace `trunc()` function to show the top ten VM events captured during the sampling period.

```
solaris# ./vmtop10.d
EXEC             FUNCTION         NAME             COUNT
tnslsnr          as_fault         prot_fault       687
arch             as_fault         as_fault         1014
oracle           page_reclaim     pgfrec           1534
oracle           page_reclaim     pgrec            1534
oracle           anon_private     cow_fault        1555
tnslsnr          as_fault         as_fault         1598
java             anon_zero        zfod             6354
oracle           anon_zero        zfod             6382
```

```
java             as_fault         as_fault         6489
m2loader         as_fault         as_fault         7332
oracle           as_fault         as_fault         16443

EXEC             FUNCTION         NAME             COUNT
java             page_reclaim     pgrec            49
run_m3loader     as_fault         as_fault         58
arch             anon_private     cow_fault        60
uname            as_fault         as_fault         120
tds_job_status   as_fault         as_fault         134
arch             as_fault         as_fault         229
java             anon_zero        zfod             2778
java             as_fault         as_fault         2907
oracle           anon_zero        zfod             3904
oracle           as_fault         as_fault         4512
[...]
```

The output generated by the `vmtop10.d` script allows us to see which processes are generating which VM events. Since we are aggregating on process name (`execname`), all processes with the same name (for example, `oracle`) will show up as one line for a given `probefunc`/`probename` pair. A predicate can be added to the `vm.d` script to just take action for `oracle` processes (`/execname == "oracle"/`), and `execname` could be replaced with `pid` as an aggregation key.

It is often interesting to examine just a specific area of the VM system, such as page-in operations. Page-ins are interesting because they represent disk I/O (read) and thus are heavier-weight operations than many other VM functions.

```
solaris# dtrace -qn 'vminfo:genunix:pageio_setup:*pgin
    { @[execname,probename] = count(); }
    END { printa("%-12s %-12s %-@12d\n",@); }'
^C
zsched       fspgin       75
zsched       pgin         75
zsched       pgpgin       75
m2loader     fspgin       686
m2loader     pgin         686
m2loader     pgpgin       686
```

Here's an example of drilling down on the `m2loader` process to get an idea of the page-in activity:

```
solaris# dtrace -qn 'vminfo:genunix:pageio_setup:*pgin
/ execname == "m2loader" / { @[stack(),ustack()] = count(); }'
^C

. . .

              nfs`nfs4_getapage+0x1c1
              nfs`nfs4_getpage+0xe2
              genunix`fop_getpage+0x47
```

continues

```
              genunix`segvn_fault+0x8b0
              genunix`as_fault+0x205
              unix`pagefault+0x8b
              unix`trap+0x3d7
              unix`_cmntrap+0x140

              libmysqlclient.so.16.0.0`adler32+0x685
              libmysqlclient.so.16.0.0`read_buf+0x62
              libmysqlclient.so.16.0.0`fill_window+0x969
              libmysqlclient.so.16.0.0`deflate_slow+0x1ff
              libmysqlclient.so.16.0.0`deflate+0x82d
              m2loader`read_zstream+0x1a3
              m2loader`get_filecont+0x2a0
              m2loader`store_files_data+0xb9c
              m2loader`main+0x463
              m2loader`0x403b3c
              198

              nfs`nfs4_getapage+0x1c1
              nfs`nfs4_getpage+0xe2
              genunix`fop_getpage+0x47
              genunix`segvn_fault+0x8b0
              genunix`as_fault+0x205
              unix`pagefault+0x8b
              unix`trap+0x3d7
              unix`_cmntrap+0x140

              libmysqlclient.so.16.0.0`adler32+0x685
              libmysqlclient.so.16.0.0`read_buf+0x62
              libmysqlclient.so.16.0.0`fill_window+0x969
              libmysqlclient.so.16.0.0`deflate_slow+0x1ff
              libmysqlclient.so.16.0.0`deflate+0x82d
              m2loader`read_zstream+0x1a3
              m2loader`get_filecont+0x2a0
              m2loader`store_files_data+0xb9c
              m2loader`main+0x463
              m2loader`0x403b3c
              225

              nfs`nfs4_getapage+0x1c1
              nfs`nfs4_getpage+0xe2
              genunix`fop_getpage+0x47
              genunix`segvn_fault+0x8b0
              genunix`as_fault+0x205
              unix`pagefault+0x8b
              unix`trap+0x3d7
              unix`_cmntrap+0x140

              libmysqlclient.so.16.0.0`adler32+0x66
              libmysqlclient.so.16.0.0`read_buf+0x62
              libmysqlclient.so.16.0.0`fill_window+0x969
              libmysqlclient.so.16.0.0`deflate_slow+0x1ff
              libmysqlclient.so.16.0.0`deflate+0x82d
              m2loader`read_zstream+0x1a3
              m2loader`get_filecont+0x2a0
              m2loader`store_files_data+0xb9c
              m2loader`main+0x463
              m2loader`0x403b3c
              354
```

The previous example shows an interesting approach: combining both the `stack()` (kernel stack) and `ustack()` (user stack) DTrace functions as aggregation keys. It allows us to see the code path from the user process up through the kernel. In this example, we can see that the user code is executing an internal read function (`read_zstream()`), which calls into the `libmysqlclient.so` library. In the user library, a `read_buf()` call is executed. Looking at the corresponding kernel stack, we see this causes a page fault trap that is resolved by reading a page from an NFS-mounted file system. We now have a kernel function we can measure directly related to the page fault activity of an application process. Specifically, the kernel `nfs4_getapage()` function appears at the top of the kernel stack frame collected when the `vminfo *pgin` (page-in) probes fired; thus, we can conclude that the page-ins are being handled (in this case) by `nfs4_getapage()`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt:nfs:nfs4_getapage:entry
/execname == "m2loader"/
{
        self->st = timestamp;
        @calls = count();
}
fbt:nfs:nfs4_getapage:return
/self->st/
{
        @mint = min(timestamp - self->st);
        @maxt = max(timestamp - self->st);
        @avgt = avg(timestamp - self->st);
        @t[“ns”] = quantize(timestamp - self->st);
        self->st = 0;
}
END
{
        normalize(@mint, 1000);
        normalize(@maxt, 1000);
        normalize(@avgt, 1000);
        printf("%-8s %-8s %-8s %-8s\n","CALLS","MIN(us)", "MAX(us)", "AVG(us)");
        printa("%-@8d %-@8d %-@8d %-@8d\n", @calls, @mint, @maxt, @avgt);
        printf("\n");
        printa(@t);
}
```

Script nfs.d

Here's an example of using several functions to measure the `nfs4_getapage()` time. We grab a time stamp at the function entry, and on the return we track the minimum, maximum, and average times, along with a quantize plot to break them down. We also track the number of calls during the sampling interval in the entry probe.

```
solaris# ./nfs.d
^C
CALLS    MIN(us)  MAX(us)  AVG(us)
79742    1        21454    42


     ns
           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@                                    9211
            2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              54209
            4096 |@@@                                      6915
            8192 |@@                                       3808
           16384 |                                         608
           32768 |                                         54
           65536 |                                         76
          131072 |@                                        1196
          262144 |@                                        1662
          524288 |@                                        1038
         1048576 |                                         847
         2097152 |                                         107
         4194304 |                                         6
         8388608 |                                         3
        16777216 |                                         2
        33554432 |                                         0
```

Note that the `nfs.d` script adjusts the `MIN`, `MAX`, and `AVG` times to microseconds, whereas the left column of the quantize aggregation is in nanoseconds (the default). We can see from the quantize distribution that most of the `getapage` operations are in the 1-microsecond to 4-microsecond range, with some outliers in the 16-microsecond to 32-millisecond range, which aligns with the `MAX` value of 21.4 milliseconds. Interesting to note with this data is the `AVG` value of 42 microseconds, which falls right around the middle of the values in the quantize graph.

One of the key points to take away from this example is the method applied to determine *what* can be measured. By starting with some basic event monitoring and drilling down with stack traces, we can see which functions are interesting to measure to understand the latency component of these operations. We should also point out that, in Solaris, the VM's page-in/page-out facilities are invoked for most file system read/write operations (that is, they are not just a sign of paging due to low memory).

Additional information on the time spent waiting for page faults can be measured as follows:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN { trace("Tracing...Ouput after 10 seconds, or Ctrl-C\n"); }

```

```
fbt:unix:pagefault:entry
{
        @st[execname] = count();
        self->pfst = timestamp
}
fbt:unix:pagefault:return
/self->pfst/
{
        @pft[execname] = sum(timestamp - self->pfst);
        self->pfst = 0;
}
tick-10s
{
        printf("Pagefault counts by execname ...\n");
        printa(@st);

        printf("\nPagefault times(ns) by execname...\n");
        printa(@pft);

        trunc(@st);
        trunc(@pft);
}
```

Script pf.d

The `pf.d` script uses the fbt provider to instrument the kernel `pagefault()` function, grabbing counts at the entry point and using the `timestamp` built-in variable and `sum()` function to total the amount of time on a per-`execname` basis. The use of a ten-second interval is a trade-off between sampling too frequently and managing a lot of output vs. sampling too infrequently and accumulating large values. Change this value to whatever interval suits your needs.

```
solaris# ./pf.d
Pagefault counts by execname ...

  httpd                                                            2
  dtrace                                                          89
  rm                                                             112
  run_m2loader                                                   164
  getopt                                                         245
  date                                                           256
  cat                                                            348
  ypserv                                                         557
  sge_execd                                                      563
  m2loader                                                       797
  sge_shepherd                                                   932
  isainfo                                                       1185
  run_m3loader                                                  1517
  tds_job_status                                                1594
  uname                                                         2300
  arch                                                          5037
  tnslsnr                                                      18728
  oracle                                                       94745
  java                                                        173492

Pagefault times(ns) by execname...
```

continues

```
  httpd                                                         73166
  dtrace                                                       816878
  rm                                                          1175890
  run_m2loader                                                2027402
  date                                                        2913004
  getopt                                                      3130660
  cat                                                         3817559
  ypserv                                                      6227314
  sge_execd                                                   7472141
  sge_shepherd                                               11802223
  isainfo                                                    15452818
  tds_job_status                                             19890730
  run_m3loader                                               20775753
  uname                                                      27416443
  arch                                                       56803348
  m2loader                                                   61545883
  tnslsnr                                                   246221638
  oracle                                                   1451181619
  java                                                     5741000093
[...]
```

Note that the interval in the `pf.d` script is ten seconds, and the times reported are in nanoseconds. In the last two lines of the sample output, we see that all the `java` processes combined spent 5.7 seconds in the 10-second sampling period wait on page faults. All the `oracle` processes combined spent 1.45 seconds. Drilling down to the underlying cause of the page faults is more challenging. A page fault occurs when a process (thread) references a virtual address in its address space that does not have a corresponding physical address—an actual mapping to a physical memory page does not exist. The page may be in memory, in which case the kernel only needs to set up the mapping (minor page fault), or the page may not be in memory, requiring a physical disk I/O (major page fault). In a modern operating system, memory is demand-paged, so the occurrence of page faults is not necessarily indicative of a problem.

Another area of process memory growth is stack space. In Solaris, the kernel `grow()` function is used to allocate space for stack growth.

```
solaris# dtrace -n 'fbt::grow:entry { @s[execname] = count(); }'
dtrace: description 'fbt::grow:entry ' matched 1 probe
^C

  m2loader                                                          1
  sleep                                                             1
  ls                                                                2
  msg-watch                                                         2
  run_m2loader                                                      2
  sge_execd                                                         2
  rm                                                                3
  date                                                              5
  getopt                                                            5
  sge_shepherd                                                      8
  tnslsnr                                                           8
```

```
  cat                                                          9
  run_extractor                                               10
  run_m3loader                                                10
  isainfo                                                     25
  java                                                        25
  qconf                                                       27
  tds_job_status                                              30
  uname                                                       75
  arch                                                        85
  oracle                                                      94

solaris# dtrace -n 'fbt::grow:entry { @s[stack()] = count(); }'
dtrace: description 'fbt::grow:entry ' matched 1 probe
^C
[...]
              unix`trap+0x1250
              unix`_cmntrap+0x140
              unix`suword64+0x21
              genunix`stk_copyout+0x72
              genunix`exec_args+0x309
              elfexec`elfexec+0x3db
              genunix`gexec+0x218
              genunix`exec_common+0x917
              genunix`exece+0xb
              unix`sys_syscall+0x17b
               15

              unix`trap+0x1250
              unix`_cmntrap+0x140
              unix`suword32+0x21
              genunix`stk_copyout+0x72
              genunix`exec_args+0x309
              elfexec`elf32exec+0x3db
              genunix`gexec+0x218
              genunix`exec_common+0x917
              genunix`exece+0xb
              unix`sys_syscall32+0x101
               31

              unix`trap+0x13c6
              unix`_cmntrap+0x140
              201
```

The two examples shown previously show the use of the same probe, but aggregating on different data. First, we aggregate on the process name (execname), and second we capture kernel stack frames, which can be useful in understanding the underlying source of events of interest. In this example, we can see several of the calls to `grow()` originate from exec systems calls, so there's some process creation happening with this workload. Most of the `grow()` calls originated as a result of a system trap, which is not at all unusual. The page fault trap handler will call `grow()` to increase a stack segment.

Kernel Memory

Tracking kernel memory is generally a complex process, for several reasons:

- Solaris, Mac OS X, and FreeBSD all implement sophisticated kernel memory allocation subsystems that leverage object caching mechanisms and reuse.[5]
- The tools available for observing kernel memory are cryptic in nature and require some knowledge of the kernel and the internals of the allocation mechanisms in order to use them effectively.
- The DTrace provider required to track kernel memory allocations is fbt, which, as we have pointed out several times, is an unstable provider.
- Using the fbt provider with DTrace to track kernel memory requires knowledge of the internals of the kernel and kernel memory allocation subsystem.

Having shared those caveats and given the complexity inherent in observing kernel memory allocation and use, we will provide some methods that can be used if there is a need to troubleshoot issues related to kernel memory. Because this is not an operating systems internals text, we must make some assumptions about your knowledge of kernel internals or your willingness to do research in source code and books. Additional methods for using DTrace to track kernel memory allocation are covered in Chapter 12.

In addition to DTrace, Solaris offers several tools for observing kernel memory:

- The `mdb(1) memstat` dcmd (shown previously)
- The `mdb(1) kmastat` dcmd
- The `kstat(1)` command

Here's an example of monitoring Solaris kernel object allocations using DTrace, aggregating on the names of the internal caches, and using the `sum()` function to track the volume of allocation requests while tracing:

```
solaris# dtrace -n 'fbt::kmem_cache_alloc:entry
{ @[args[0]->cache_name] = sum(args[0]->cache_bufsize); }'
dtrace: description 'fbt::kmem_cache_alloc:entry ' matched 1 probe
^C
[...]
```

5. See "The Slab Allocator: An Object-Caching Kernel Memory Allocator" by Jeff Bonwick, and "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources" by Jeff Bonwick and Jonathan Adams.

```
vn_cache                                                    1469760
streams_dblk_80                                             1803648
streams_dblk_208                                            1859520
HatHash                                                     3276800
zio_buf_131072                                              3407872
anon_cache                                                  3625920
kmem_alloc_1152                                             3867264
kmem_alloc_192                                              4345344
hment_t                                                     5134272
streams_dblk_20304                                          5267328
kmem_alloc_32                                               5487296
kmem_alloc_64                                               9930112
zio_cache                                                 100685360
kmem_alloc_4096                                           269684736
```

Given that the most frequently used cache in this sample was a generic `kmem_alloc_4096` cache, we can use a predicate and kernel stack trace to determine which kernel facilities are generating these allocations:

```
solaris# dtrace -n 'fbt::kmem_cache_alloc:entry
/args[0]->cache_name == "kmem_alloc_4096"/ { @[stack()] = count(); }'
dtrace: description 'fbt::kmem_cache_alloc:entry ' matched 1 probe
^C
[...]
              genunix`kmem_alloc+0x70
              genunix`exec_args+0xe5
              elfexec`elf32exec+0x3db
              genunix`gexec+0x218
              genunix`exec_common+0x917
              genunix`exece+0xb
              unix`sys_syscall32+0x101
              144

              genunix`kmem_zalloc+0x3b
              genunix`anon_set_ptr+0xc9
              genunix`anon_dup+0x83
              genunix`segvn_dup+0x51c
              genunix`as_dup+0xf8
              genunix`cfork+0x661
              genunix`fork1+0x10
              unix`sys_syscall+0x17b
              257

              genunix`kmem_alloc+0x70
              genunix`segvn_fault_anonpages+0x177
              genunix`segvn_fault+0x23a
              genunix`as_fault+0x205
              unix`pagefault+0x8b
              unix`trap+0x3d7
              unix`_cmntrap+0x140
            59978
```

The kernel stack frames captured for `kmem_alloc()` from the `kmem_alloc_4096` cache indicate most of those allocation requests are the direct result of page faults, generated by user processes. Recall that we can instrument the `pagefault` routine to see which user processes are incurring page faults:

```
solaris# dtrace -n 'fbt:unix:pagefault:entry { @[execname] = count(); }'
dtrace: description 'fbt:unix:pagefault:entry ' matched 1 probe
^C
[...]
  tds_job_status                                                  2380
  arch                                                            3778
  tnslsnr                                                         5909
  m2loader                                                        8581
  oracle                                                         35603
  java                                                           53396
```

Reference the "Memory" section, which covers user memory and drilling down on page faults from a user process perspective.

Note that calls into the `kmem_alloc()` family of routines in Solaris do not necessarily mean physical memory is being allocated. Most of the time, the kernel is taking advantage of the design features of the slab allocator and simply reusing memory from its object caches.

The Solaris kernel implements a vmem layer as a universal backing store for the kmem caches and general-purpose kernel resource allocation. The vmem layer can be observed using a very similar set of DTrace programs.

```
solaris# dtrace -n 'fbt::vmem_alloc:entry { @[args[0]->vm_name] = sum(arg1); }'
dtrace: description 'fbt::vmem_alloc:entry ' matched 1 probe
^C

  crypto                                                             2
  tl_minor_space                                                     2
  contracts                                                          3
  ip_minor_arena_la                                                 47
  ip_minor_arena_sa                                                 79
  bp_map                                                       1040384
  kmem_oversize                                                1314320
  kmem_firewall_va                                             1343488
  segkp                                                        1400832
  kmem_io_4G                                                  20303872
  heap                                                        26734592
```

We see the largest vmem arenas during the sampling period are `heap` and `kmem_io_4G`. As before, we can use these names in predicates to capture kernel stack frames of interest.

```
solaris# dtrace -n 'fbt::vmem_alloc:entry
    /args[0]->vm_name == "heap"/ { @[stack()] = count(); }'
dtrace: description 'fbt::vmem_alloc:entry ' matched 1 probe
^C
[…]
              unix`segkmem_xalloc+0x144
              unix`segkmem_alloc_io_4G+0x26
              genunix`vmem_xalloc+0x315
              genunix`vmem_alloc+0x155
```

```
                unix`kalloca+0x160
                unix`i_ddi_mem_alloc+0xd6
                rootnex`rootnex_setup_copybuf+0xe4
                rootnex`rootnex_bind_slowpath+0x2dd
                rootnex`rootnex_coredma_bindhdl+0x16c
                rootnex`rootnex_dma_bindhdl+0x1a
                genunix`ddi_dma_buf_bind_handle+0xb0
                sata`sata_dma_buf_setup+0x4b9
                sata`sata_scsi_init_pkt+0x1f5
                scsi`scsi_init_pkt+0x44
                sd`sd_setup_rw_pkt+0xe5
                sd`sd_initpkt_for_buf+0xa3
                sd`sd_start_cmds+0xa5
                sd`sd_core_iostart+0x87
                sd`sd_mapblockaddr_iostart+0x11a
                sd`sd_xbuf_strategy+0x46
                259

                unix`ppmapin+0x2f
                zfs`mappedread+0x84
                zfs`zfs_read+0x10e
                zfs`zfs_shim_read+0xc
                genunix`fop_read+0x31
                genunix`read+0x188
                genunix`read32+0xe
                unix`sys_syscall32+0x101
                271
```

In the previous example, monitoring the vmem heap arena, we can see allocation requests coming from the ZFS layer on behalf of read system calls and the SATA driver for DMA operations.

Another layer that can be instrumented in Solaris is the `segkmem` layer, which is the segment driver for kernel address space segments that gets called from the vmem layer. The following series of DTrace programs show a very similar flow to what we have used to look at the `kmem` and `vmem` layers:

```
solaris# dtrace -n 'fbt::segkmem*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::segkmem*:entry ' matched 41 probes
^C

  segkmem_alloc                                                    19
  segkmem_zio_alloc                                               680
  segkmem_alloc_vn                                                699
  segkmem_page_create                                             699
  segkmem_alloc_io_4G                                            3562
  segkmem_free                                                   3577
  segkmem_free_vn                                                3577
  segkmem_xalloc                                                 4261

solaris# dtrace -n 'fbt::segkmem_xalloc:entry
    { @[args[0]->vm_name,arg2] = count(); }'
dtrace: description 'fbt::segkmem_xalloc:entry ' matched 1 probe
^C
[...]
  heap                                                          16384               64
  heap                                                           8192               92
  heap                                                          12288               96
```
continues

```
  heap                                                          135168            852
solaris# dtrace -n 'fbt::segkmem_xalloc:entry
    /args[0]->vm_name == "heap"/ { @[stack()] = count(); }'
dtrace: description 'fbt::segkmem_xalloc:entry ' matched 1 probe
^C
[...]
              unix`segkmem_alloc_io_4G+0x26
              genunix`vmem_xalloc+0x315
              genunix`vmem_alloc+0x155
              unix`kalloca+0x160
              unix`i_ddi_mem_alloc+0xd6
              rootnex`rootnex_setup_copybuf+0xe4
              rootnex`rootnex_bind_slowpath+0x2dd
              rootnex`rootnex_coredma_bindhdl+0x16c
              rootnex`rootnex_dma_bindhdl+0x1a
              genunix`ddi_dma_buf_bind_handle+0xb0
              sata`sata_dma_buf_setup+0x4b9
              sata`sata_scsi_init_pkt+0x1f5
              scsi`scsi_init_pkt+0x44
              sd`sd_setup_rw_pkt+0xe5
              sd`sd_initpkt_for_buf+0xa3
              sd`sd_start_cmds+0xa5
              sd`sd_return_command+0xd7
              sd`sdintr+0x187
              sata`sata_txlt_rw_completion+0x145
              nv_sata`nv_complete_io+0x95
              369
```

The three consecutive DTrace programs executed in the previous example show, first, how to determine which `segkmem` routines are being called. We see `segkmem_xalloc()`, which certainly looks like an allocation function, and we next instrument the entry point to that function, aggregating on the name of the `vmem` arena and calling into it and the size. Finally, we take the name from the generated output (`heap`) and use it in a predicate in the third program, which captures kernel stacks. We can see `segkmem_xalloc()` getting called out of `vmem_alloc()`, entered from the SATA drive for DMA memory.

The examples shown are intended to provide some basic steps you can take to determine which kernel subsystems are making use of kernel memory allocators. There is nothing observed in the examples shown that indicates a problem.

Mac OS X offers a `zprint(1)` utility, which generates output similar to the `kmastat` dcmd available in DTrace, listing information about the many kernel object caches in use.

`kernel_memory_allocate()` is commonly (but not exclusively) used in the Mac OS X kernel for kernel memory allocation. This can be instrumented with DTrace using the fbt provider, and kernel stack traces can be captured to determine which kernel subsystem is allocating memory:

```
macosx> dtrace -n 'fbt::kernel_memory_allocate:entry
    { @[stack()] = quantize(arg2); }'
dtrace: description 'fbt::kernel_memory_allocate:entry ' matched 1 probe
```

```
^C
[...]
              mach_kernel`kmem_alloc+0x38
              mach_kernel`kalloc_canblock+0x76
              mach_kernel`OSMalloc+0x60
              0x5a5bcd93
              0x5a5be95b
              0x5a5befcc
              mach_kernel`decmpfs_hides_rsrc+0x5f3
              mach_kernel`decmpfs_pagein_compressed+0x1b6
              mach_kernel`hfs_vnop_pagein+0x64
              mach_kernel`VNOP_PAGEIN+0x9e
              mach_kernel`vnode_pagein+0x30b
              mach_kernel`vnode_pager_cluster_read+0x5c
              mach_kernel`vnode_pager_data_request+0x8a
              mach_kernel`vm_fault_page+0xcaa
              mach_kernel`vm_fault+0xd2d
              mach_kernel`user_trap+0x29f
              mach_kernel`lo_alltraps+0x12a

           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@                                  11
           16384 |@@@@@@@@@                                13
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@                 37
           65536 |                                         0


              mach_kernel`kmem_alloc+0x38
              mach_kernel`kalloc_canblock+0x76
              mach_kernel`kalloc+0x19
              mach_kernel`dt_kmem_alloc_aligned+0x1a
              mach_kernel`helper_init+0x20c
              mach_kernel`helper_ioctl+0x36d
              mach_kernel`spec_ioctl+0x9d
              mach_kernel`VNOP_IOCTL+0xdc
              mach_kernel`utf8_encodelen+0x677
              mach_kernel`fo_ioctl+0x3f
              mach_kernel`ioctl+0x519
              mach_kernel`unix_syscall+0x243
              mach_kernel`lo_unix_scall+0x118

           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              10
           16384 |                                         0
           32768 |                                         0
           65536 |@@@@@@@@@@@@@                            5
          131072 |                                         0
```

The DTrace command used for Mac OS X instruments the entry point of `kernel_memory_allocate()` and aggregates on the kernel stack using quantize to track size allocation size (`arg2`). Here we see several allocations in the 8KB to 64KB range (top stack) from the trap code—most likely page fault traps, calling into the page-in code. The bottom frame and associated aggregation shows ten allocations in the 8KB to 16KB range and five in the 64KB to 128KB range, orginating from `ioctl(2)` system calls.

FreeBSD also makes use of a `kmem` layer for object caching kernel objects in kernel memory:

```
[root@freebsd ~]# dtrace -n 'fbt::kmem*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::kmem*:entry ' matched 18 probes
^C

  kmem_alloc_wait                                                   9
  kmem_free_wakeup                                                  9
  kmem_malloc                                                    3908

[root@freebsd ~]# dtrace -n 'fbt::kmem_malloc:entry { @[stack()] = count(); }'
dtrace: description 'fbt::kmem_malloc:entry ' matched 1 probe
^C
[...]

              kernel`page_alloc+0x27
              kernel`keg_alloc_slab+0xfd
              kernel`keg_fetch_slab+0xd4
              kernel`zone_fetch_slab+0x4c
              kernel`uma_zalloc_arg+0x4ae
              kernel`getnewvnode+0x155
              kernel`ffs_vgetf+0x112
              kernel`ffs_vget+0x2e
              kernel`ufs_lookup_+0xaa9
              kernel`ufs_lookup+0x1e
              kernel`VOP_CACHEDLOOKUP_APV+0x7c
              kernel`vfs_cache_lookup+0xd6
              kernel`VOP_LOOKUP_APV+0x84
              kernel`lookup+0x70e
              kernel`namei+0x7bf
              kernel`kern_statat_vnhook+0x72
              kernel`kern_statat+0x3c
              kernel`kern_lstat+0x36
              kernel`lstat+0x2f
              kernel`syscall+0x3e5
             1235
```

When exploring a subsystem we are not familiar with, we often choose to look first at which functions of interest are getting called, in this case from the kernel `kmem` functions. We can then instrument a function of interest, `kmem_malloc()`, aggregating on kernel stacks. With the FreeBSD system shown here, we see kernel memory allocations coming up from `lstat(2)` system calls, through the file system layers and into the FreeBSD kernel slab routines.

Refer to Chapter 12 for more examples of kernel DTrace analysis.

Memory Summary

When observing memory, the metrics that interest us are used memory vs. free memory and where used memory is being used. System utilities that provide systemwide metrics and per-process metrics on memory are the right place to start. DTrace can determine where and why the kernel and applications are consuming memory and measure the latency impact of memory-related events such as page faults.

Observing Disk and Network I/O

We cover disk I/O and network I/O extensively in Chapters 4 to 7. In the interest of completeness, we'll outline some preliminary methods for understanding disk and network I/O in this chapter. We encourage you to reference the dedicated chapters for drilling down further for observing and understanding I/O.

From a system perspective, it is relatively simple to observe disk and network I/O activity using bundled tools and utilities. In Solaris, start with `netstat(1M)` (network) and `iostat(1M)` (disk). The Mac OS X and FreeBSD operating systems also include `iostat(8)` and `netstat(8)` utilities (reference the man pages for implementation and command-line argument differences).

I/O Strategy

Once an overall view of the system has been established, use DTrace as your connect-the-dots technology to do the following:

- Determine which processes and threads are generating disk and network I/O
- Determine the rate and latency of I/Os to disks and the network
- Determine the I/O latency profile of your application processes
- Track disk I/O targets (files, file systems, raw devices)

I/O Checklist

Table 3-7 provides an I/O checklist.

Table 3-7 I/O Checklist

Issue	Description
Device I/O profile	Which devices are targeted for I/O, and what is the read/write breakdown?
Device I/O latency	How fast are I/Os being processed on a per-device basis?
Device I/O errors	What device errors are occurring, and are they being handled by the OS?
Application I/O profile	Which components of the workload are generating I/O?
Application I/O latency	What is the cost of I/O latency in terms of delivered application performance?
Application I/O errors	Is the application encountering I/O errors?

I/O Providers

Table 3-8 and Figure 3-1 list the DTrace providers used most often when observing disk and network I/O.

Table 3-8 I/O Providers

Provider	Description
io	Stable provider for observing physical disk I/O (and NFS back-end I/O on Solaris)
fbt	Kernel memory modules and functions related to disk and network I/O
syscall	Application-issued systems calls for disk and network I/O
mib	Message Information Base provider, traces counters served by SNMP
ip	Stable provider for observing IP-layer network traffic
fsinfo	File system operations (Solaris)
vfs	Virtual file system provider (FreeBSD)

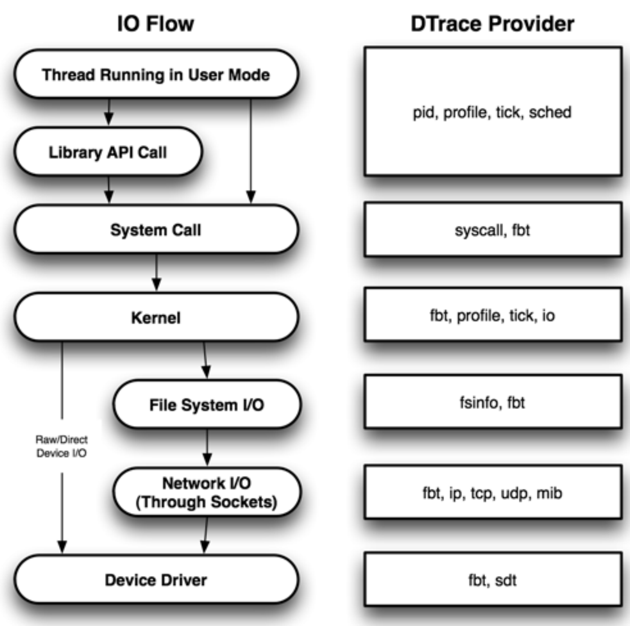


Figure 3-1 DTrace providers for I/O

Check your operating system version to see which of these providers are available. For example, the ip provider is currently available only in OpenSolaris.

I/O One-Liners

The following one-liners can be used to begin understanding the I/O load at both an application and system level.

syscall Provider

These may need adjustments to match the system calls on your operating system (for example, `syscall::*read:entry` does not match the `read_nocancel` system call on Mac OS X).

Which processes are executing common I/O system calls?

```
dtrace -n 'syscall::*read:entry,syscall::*write:entry { @rw[execname,probefunc] =
count(); }'
```

Which file system types are targeted for reads and writes?

```
dtrace -n 'syscall::*read:entry,syscall::*write:entry { @fs[execname, probefunc,
fds[arg0].fi_fs] = count(); }'
```

Which files are being read, and by which processes?

```
dtrace -n 'syscall::*read:entry { @f[execname, fds[arg0].fi_pathname] = count(); }'
```

Which files are being written, and by which processes?

```
dtrace -n 'syscall::*write:entry { @f[execname, fds[arg0].fi_pathname] = count(); }'
```

Other Providers

Which processes are generating network I/O (Solaris)?

```
dtrace -n 'fbt:sockfs::entry { @[execname, probefunc] = count(); }'
```

Which processes are generating file system I/O (Solaris)?

```
dtrace -n 'fsinfo::: { @fs[execname, probefunc] = count(); }'
```

What is the rate of disk I/O being issued?

```
dtrace -n 'io:::start { @io = count(); } tick-1sec { printa("Disk I/Os per second: %@d
\n", @io); trunc(@io); }'
```

Note also that the `/usr/demo/dtrace` directory on Solaris systems contains several useful scripts for observing and measuring I/O.

I/O Analysis

When observing and measuring I/O, it's important to be aware of the asynchronous nature of I/O and how this affects what typically happens when a thread issues a read or write to a disk, network interface, or file system file. At some point after a system call is invoked, the calling thread will be put to sleep while the I/O moves down through the kernel, through the device driver, out over the wire, and back again. Once the I/O is completed, the kernel will copy the data to the thread that executed the read or write and issue a wake-up to the thread.

As illustrated in Figure 3-2, once an application thread issues an I/O, it will be taken off the CPU until the I/O completes. The actual processing of the I/O through the kernel layers down into the device driver happens asynchronously with respect to the thread that issued the I/O. From an observability perspective, this means that when you are instrumenting lower layers of the I/O stack and want to correlate I/O events to processes and threads using DTrace variables, `execname`, `pid`, and `tid` may not provide the expected results, because the issuing thread will likely not be on the CPU when probes instrumenting the lower layers of the I/O stack fire.

With that in mind, the key components to I/O analysis are measuring I/O rates and I/O latency and determining to what extent I/O latency is affecting delivered workload performance. Much of this is covered in the dedicated chapters, but we discuss some methods that can be applied here.

All I/O, from an application/workload perspective, begins with system calls. There are several system calls that are used by applications to do disk I/O, the most common being `read(2)` and `write(2)`. Other variants include `pread(2)`, `readv(2)`, `pwrite(2)`, `writev(2)`, and so on, depending on your operating sys-

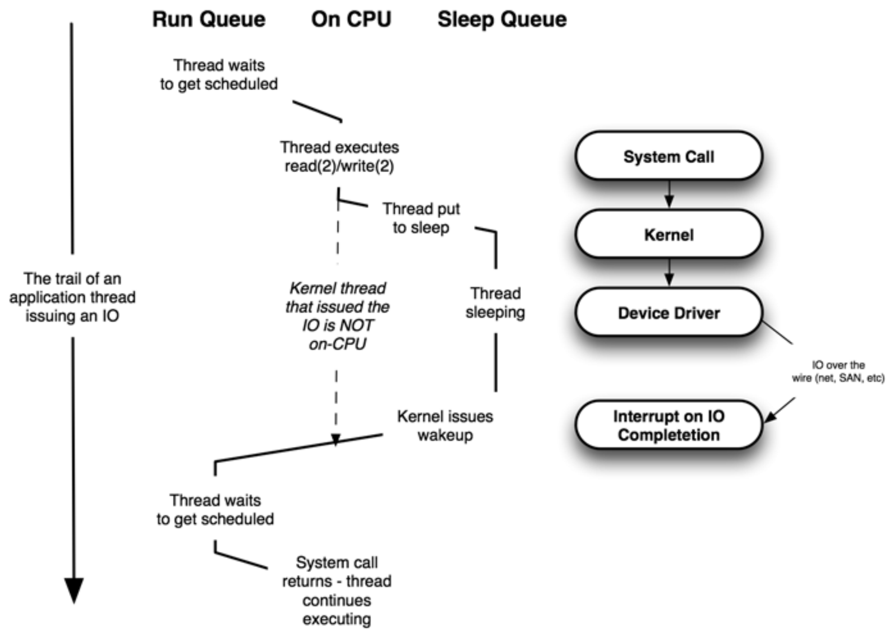


Figure 3-2 I/O flow

tem version. You can start by using the DTrace syscall provider to observe which processes are issuing which system calls and drilling down from there based on the type of I/O system call in use. This is done by the following script:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall:::entry
{
        @[execname, probefunc] = count();
}
END
{
        trunc(@, 10);
        printf("%-16s %-16s %-8s\n", "EXEC", "SYSCALL", "COUNT");
        printa("%-16s %-16s %-@8d\n",@);
}
```

Script sctop10.d

The `sctop10.d` script truncates the output so we see only the top ten system calls and calling process names.

```
solaris# ./sctop10.d
^C
EXEC                SYSCALL            COUNT
java                stat               1127
sge_shepherd        getuid             1168
arch                sigaction          1272
dtrace              ioctl              1599
m2loader            brk                1606
arch                read               1808
sge_shepherd        close              2152
sge_execd           close              2197
java                lseek              6183
java                read               6388

macosx# ./sctop10.d
^C
EXEC                SYSCALL            COUNT
VBoxXPCOMIPCD       sendto             112
VirtualBoxVM        recvfrom           112
WindowServer        sigaltstack        118
WindowServer        sigprocmask        118
VBoxSVC             select             125
VBoxSVC             __semwait_signal   136
Mail                __semwait_signal   147
VirtualBoxVM        select             148
VirtualBoxVM        __semwait_signal   942
VirtualBoxVM        ioctl              68972

[root@freebsd /var/tmp]# ./sctop10.d
^C
EXEC                SYSCALL            COUNT
ls                  fchdir             1643
ls                  close              1654
ls                  open               1657
ls                  fstat              2201
ls                  lstat              8062
sshd                write              8491
sshd                read               8501
ls                  write              9295
sshd                select             16983
sshd                sigprocmask        33966
```

The previous three examples show a big-picture system call profile from Solaris, OS X, and FreeBSD. On the Solaris system, we can see several processes executing `read(2)`, whereas on the Mac OS X system we appear to have mostly network I/O (`sendto` and `recvfrom` system calls). The FreeBSD system was executing some `ls(1)` commands, and the `sshd` daemon was doing some reads and writes.

By making some changes to the `sctop10.d` script, we can make it much more I/O centric and learn more about the processes generating I/O on the system.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall::*read*:entry,
syscall::*write*:entry
{
```

```
8          @[execname, probefunc, fds[arg0].fi_fs] = count();
9  }
10 END
11 {
12         trunc(@, 10);
13         printf("%-16s %-16s %-8s %-8s\n", "EXEC", "SYSCALL", "FS", "COUNT");
14         printa("%-16s %-16s %-8s %-@8d\n",@);
15 }
```

Script scrwtop10.d

The `scrwtop10.d` script enables syscall provider probes for various read/write system calls, using the * pattern matching character in the probefunc field. As noted previously, this may result in matches on system calls that are not of interest,[6] so you should verify which system calls will be instrumented on your target host by running `dtrace -ln 'syscall::*read*:entry'` and do the same for the write probe.

Note that the use of the `fds[]` array as an aggregation key requires that the probes specified take a file descriptor as the first argument (line 8, `arg0`). For FreeBSD, which has not yet implemented the `fds[]` array, the key can be changed to simply `arg0`, with changes to the output header to reflect the field is a file descriptor (FD), not a file system (FS, line 13). Once again, we make use of the `trunc()` function to generate just the top ten events captured during tracing.

```
solaris# ./scrwtop10.d
^C
EXEC             SYSCALL          FS       COUNT
Xvnc             read             sockfs   671
oracle           pwrite           zfs      1080
arch             read             lofs     1188
mysqld           read             sockfs   1385
oracle           write            sockfs   2295
oracle           read             sockfs   2322
java             write            nfs4     4538
java             read             sockfs   5630
java             read             zfs      15703
java             read             lofs     29359
```

Here we get a better view of the I/O target by observing which file system layer is being used, giving us more insight into the nature of the I/O load on the system. Here's another view, using the fsinfo provider:

```
#!/usr/sbin/dtrace -qs

```

continues

6. An example is `readlink` on Solaris.

```
3  fsinfo:::
4  {
5          @[execname,probefunc] = count();
6  }
7  END
8  {
9          trunc(@,10);
10         printf("%-16s %-16s %-8s\n","EXEC","FS FUNC","COUNT");
11         printa("%-16s %-16s %-@8d\n",@);
12 }
```

Script fstop10.d

As you can see, a DTrace script easily becomes a template on which to build other scripts with minor changes. Things such as output formatting in `END` clauses are easily reused, and minor edits to aggregations keys or the addition of predicates to drill down further can be done quickly and easily.

In the `fs.d` script, we leverage the Solaris fsinfo provider to gain insight on I/O based on the functions called in the file system–independent layer of the kernel. The `probefunc` used here as an aggregation key (line 5) provides the name of the kernel function that can be used in subsequent scripts for drill down if necessary. `probename` can be used instead of `probefunc` to provide a generic FS operation name that will remain stable across operating systems and releases (see `fstop10_enhanced.d`, later in this section).

```
solaris# ./fstop10.d
^C
EXEC             FS FUNC          COUNT
oracle           fop_write        14494
java             fop_readlink     16410
java             fop_seek         28161
oracle           fop_rwlock       28551
oracle           fop_rwunlock     28613
java             fop_access       32845
java             fop_read         59105
java             fop_rwlock       60449
java             fop_rwunlock     60449
java             fop_lookup       110724
```

The output produced by `fstop10.d` shows us that, during this sampling period, `java` processes generated a large number of I/O calls through the kernel's virtual file system layer. Note that these operations are not necessarily on-disk file systems (like ZFS or UFS) but may be I/Os to `sockfs` (network), `devfs` (devices), and so on. We can enhance our view with more detail using the fsinfo provider by taking advantage of available arguments. `args[0]` is a pointer to a `fileinfo_t` structure, and `args[1]` is the return value from the file system operation associated with the probe. A return value of zero means success. Other return values depend on the actual file system operation; for example, the return for fop_read() and fop_write() is the number of bytes read or written. For reference, the fileinfo_t structure contains the following members:

```
typedef struct fileinfo {
        string fi_name;                 /* name (basename of fi_pathname) */
        string fi_dirname;              /* directory (dirname of fi_pathname) */
        string fi_pathname;             /* full pathname */
        offset_t fi_offset;             /* offset within file */
        string fi_fs;                   /* filesystem */
        string fi_mount;                /* mount point of file system */
} fileinfo_t;
```

Here's a modified version of the fstop10.d script:

```
#!/usr/sbin/dtrace -qs

fsinfo:::
{
        @[execname,probename,args[0]->fi_fs,args[0]->fi_pathname] = count();
}
END
{
        trunc(@,10);
        printf("%-16s %-8s %-8s %-32s %-8s\n",
         "EXEC","FS FUNC","FS TYPE","PATH","COUNT");
        printa("%-16s %-8s %-8s %-32s %-@8d\n",@);
}
```

Script fstop10_enhanced.d

We added several fields as aggregation keys, enabling us to observe the specific file system operation (probename), the file system type, and the full path name to the file, in addition to again using trunc() to display only the top ten events captured.

```
solaris# ./fstop10_enhanced.d
^C
EXEC             FS FUNC  FS TYPE  PATH                             COUNT
java             lookup   ufs      /var                             39
java             lookup   ufs      /var/webconsole                  39
java             lookup   ufs      /var/webconsole/domains          39
java             lookup   ufs      /var/webconsole/domains/console  39
java             lookup   ufs      /usr/share/webconsole/webapps    70
java             lookup   ufs      /usr                             88
java             lookup   ufs      /usr/share                       88
java             lookup   ufs      /usr/share/webconsole            88
fsflush          inactive tmpfs    /tmp/out1                        1706
fsflush          putpage  tmpfs    /tmp/out1                        1706
```

With the enhanced version of the `fs.d` script, we have a more detailed view of file I/O operations systemwide and can drill down from here based on what we observe and the problem under investigation.

Disk I/O

It may be easiest to start looking at disk I/O with bundled tools such as `iostat(1M)`, which gives a systemwide view showing key disk I/O statistics on a per-device and per-controller basis. Then use the DTrace io provider to examine details of the I/O events and, if possible, identify the processes that are generating disk I/O. The following example shows a series of DTrace programs executed from the command line, illustrating the drill-down flow and how quickly you can understand a great deal about the disk I/O load on a system.

The following examples were demonstrated on Solaris with UFS as the file system:

```
solaris# dtrace -n 'io:::start { @[execname, pid] = count(); }'
dtrace: description 'io:::start ' matched 6 probes
^C

  sched                                                         0            3
  java                                                       7453            4
  fsflush                                                       3            5
  java                                                       7480        12933
  java                                                       7486        13007
  java                                                       7495        13009
  java                                                       7498        13060
  java                                                       7492        13153
  java                                                       7489        13316
  java                                                       7483        13380
  java                                                       7500        13456

solaris# dtrace -qn 'syscall:::entry /execname == "java"/
    { @[pid, probefunc] = count(); } END { trunc(@, 10); printa(@); }'
^C

     7492  pread64                                                          6764
     7489  pread64                                                          6789
     7483  pwrite64                                                         6791
     7492  pwrite64                                                         6825
     7486  pwrite64                                                         6839
     7489  pwrite64                                                         6855
     7480  pwrite64                                                         6889
     7480  pread64                                                          6900
     7453  read                                                            23526
     7453  pollsys                                                         23644

solaris# dtrace -n 'syscall::pread*:entry,syscall::pwrite*:entry
     /execname == "java"/ { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::pread*:entry,syscall::pwrite*:entry ' matched 4 probes
^C
```

```
  specfs                                                          147582

solaris# dtrace -n 'syscall::pread*:entry,syscall::pwrite*:entry /execname == "java"/
      { @[fds[arg0].fi_pathname] = count(); }'
dtrace: description 'syscall::pread*:entry,syscall::pwrite*:entry ' matched 4 probes
^C

  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,19:c  5235
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,d:    7077
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,e:c   7250
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,a:c   7648
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,c:c   7858
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,b:c   8872
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,1a:c  8991
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,14:c 10417
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,1b:c 10674
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,18:c 10721
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,1c:c 10987
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,16:c 11859
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,17:c 12191
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,15:c 12251
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,f:c  12310
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,10:c 12483
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,12:c 12521
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,13:c 12621
  /devices/pci@0,600000/pci@0/pci@9/SUNW,emlxs@0/fp@0,0/ssd@w5000097208140919,11:c 12867
```

This shows four consecutive invocations of DTrace used to get a handle on the disk I/O load on the system. The first command uses the io provider, aggregating on process name and PID to determine which processes are generating disk I/O—on the assumption that the requesting process is still on-CPU (which may not be the case, depending on the type of I/O and file system). We see from the output that several `java` processes are generating disk I/O. The next step is to determine which system calls the `java` processes are using to do I/O. From the second command, we can see extensive use of the `pread64(2)` and `pwrite64(2)` system calls, so we follow up by using DTrace to instrument just those system calls and taking a look at the target file system.

The resulting output shows that all the pread and pwrite calls are hitting specfs, which is used in Solaris for raw or block device I/O. The last command aggregates on the file path names, and we see that the I/O targets are in fact block device files.

This is all good and useful information, but in order to properly characterize the load on the system and the application, we need to generate a few key performance metrics.

I/O rate: What is the rate of reads and writes per second?

I/O throughput: What is the data rate of reads and writes?

I/O latency: How long are the disk reads and writes taking?

```
1  #!/usr/sbin/dtrace -Cs
2
3  #pragma D option quiet
4
5  #define PRINT_HDR printf("%-8s %-16s %-8s %-16s\n","RPS","RD BYTES","WPS","WR BYTES");
6
7  dtrace:::BEGIN
8  {
9          PRINT_HDR
10 }
11
12 io:::start
13 /execname == $$1 && args[0]->b_flags & B_READ/
14 {
15         @rps = count();
16         @rbytes = sum(args[0]->b_bcount);
17 }
18
19 io:::start
20 /execname == $$1 && args[0]->b_flags & B_WRITE/
21 {
22         @wps = count();
23         @wbytes = sum(args[0]->b_bcount);
24 }
25 tick-1sec
26 {
27         printa("%-@8d %-@16d %-@8d %-@16d\n", @rps, @rbytes, @wps, @wbytes);
28         trunc(@rps); trunc(@rbytes); trunc(@wps); trunc(@wbytes);
29 }
30 tick-1sec
31 /x++ == 20/
32 {
33         PRINT_HDR
34         x = 0;
35 }
```

Script disk_io.d

The `disk_io.d` script enables two `io:::start` probes, using a predicate to separate reads from writes and testing for our process name of interest (passed as a command-line argument in the following example, lines 13 and 20). We also show the use of some time-saving features of the D language that can be integrated into scripts. We defined a header to label our output (line 5). The D language supports use of the `#define` directive, which we use here to define a macro to print the header. This makes it easier to print the header in multiple places (lines 9 and 33), as well as simplifying changes to the header. We need only edit line 5 for script modification to the header. Note that, in order for the `#define` to work, we need to instruct DTrace to invoke the C compiler preprocessor, which is done using the `-C` flag (line 1).

If the target system does not have a C preprocessor available, an alternate method of accomplishing the same thing is shown next. We set two integer variables in the `dtrace:::BEGIN` probe (`LINES`, `line`) and use a different predicate in a `tick-1sec` probe to print the header every 20 lines. The header will also be printed initially, since `line` is initialized to zero.

```
#!/usr/sbin/dtrace -s
[...]
dtrace:::BEGIN
{
   LINES = 20; line = 0;
}
[...]
tick-1sec
/--line <= 0/
{
   printf("%-8s %-16s %-8s %-16s\n", "RPS", "RD BYTES", "WPS", "WR BYTES");
   line = LINES;
}
```

We now have a script that will give us disk I/O load data for a process of interest:

```
solaris# ./disk_io.d java
RPS      RD BYTES         WPS      WR BYTES
6112     50069504         9363     76701696
5873     48111616         9482     77676544
5920     48496640         9303     76210176
5943     48685056         9345     76554240
5939     48652288         9210     75448320
5885     48209920         9264     75890688
6045     49520640         9192     75300864
5975     48947200         9415     77127680
5973     48930816         9305     76226560
^C
4808     39387136         7583     62119936
```

We can see the `java` processes are doing about 6KB reads and just more than 9KB writes per second to disk, with about 48MB/sec read throughput and 77MB/sec write throughput. With the I/O rate and throughput numbers in hand, the remaining metric of interest is latency or how long disk I/Os are taking. The `/usr/demo/dtrace` directory on Solaris systems includes a DTrace script, `iotime.d`,[7] which will provide per-I/O, per-device I/O times.

```
solaris# dtrace -s /usr/demo/dtrace/iotime.d
[...]
    ssd149                                                        <none>  W  30.899
    ssd147                                                        <none>  W  48.833
```

continues

7. The `iotime.d` script is also listed in the DTrace Guide's io provider chapter (a modified version in included in this book on the following page, `iotimeq.d`).

```
ssd155                                                      <none>  R  29.113
ssd158                                                      <none>  W  14.520
ssd148                                                      <none>  R  17.496
ssd147                                                      <none>  R  43.884
ssd156                                                      <none>  R  23.689
ssd148                                                      <none>  R   9.485
ssd155                                                      <none>  R  23.559
ssd145                                                      <none>  R  15.707
ssd151                                                      <none>  R  14.037
ssd158                                                      <none>  W   5.441
ssd161                                                      <none>  R  20.636
```

The iotime.d script provides the short device name, file path (<none> in this case because the load is block device I/O), whether the I/O was a read or write, and the time in milliseconds from I/O request to completion. This script will generate a tremendous amount of output if there is a steady rate of disk I/O traffic and many disk devices handling I/Os. We can modify the script to use an aggregation and grab a snapshot to track per-device I/O times.

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  dtrace:::BEGIN { trace(“Tracing...Output afer 10 seconds, or Ctrl-C\n”); }
6
7  io:::start
8  {
9          start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
10 }
11
12 io:::done
13 /start[args[0]->b_edev, args[0]->b_blkno]/
14 {
15         this->elapsed =
16          (timestamp - start[args[0]->b_edev, args[0]->b_blkno]) / 1000000;
17         @iot[args[1]->dev_statname,
18          args[0]->b_flags & B_READ ? "READS(ms)" : "WRITES(ms)"] =
19                quantize(this->elapsed);
20         start[args[0]->b_edev, args[0]->b_blkno] = 0;
21 }
22 tick-10sec
23 {
24         printa(@iot);
25         exit(0);
26 }
```

Script iotimeq.d

The iotimeq.d script is based on iotime.d, but instead of printing a line of output for every I/O, it uses a quantize() aggregation (line 19) to capture I/O times per-device and per I/O type (read or write). The elapsed I/O time is converted to milliseconds and stored in a clause-local variable (lines 15, 16), which is passed to the quantize function.

```
solaris# ./iotimeq.d
. . .
  ssd153                                      READS(ms)
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |                                         4
               2 |                                         10
               4 |@@                                       146
               8 |@@@@@@@@@@@                              889
              16 |@@@@@@@@@@@@@@@@@@@@@@@@                 1946
              32 |@@@@                                     338
              64 |                                         0
. . .
  ssd148                                      WRITES(ms)
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         5
               4 |@                                        75
               8 |@@@@                                     368
              16 |@@@@@@@@@@@@@@@@                         1336
              32 |@@@@@@@@@@@@@@@@@                        1424
              64 |@                                        97
             128 |                                         0
. . .
```

The previous truncated sample output shows the quantize aggregation for reads on device ssd153 and writes on device ssd148. We can see the I/O time falls mostly in the 8-millisecond to 31-millisecond range, with some writes on ssd148 approaching 128 milliseconds (which is really slow).

Another approach to measuring I/O latency is to trace file I/O at the system call layer. One advantage of this approach is that the process responsible is guaranteed to still be on-CPU and can be matched with the `execname` and `pid` built-in D variables. This is not the case with the io provider, and for some file systems such as ZFS, disk I/O is often requested by another kernel thread. This means some of the previous io provider examples that matched on `execname` will miss most ZFS disk I/O events, since the `execname` is `sched` (the kernel).

The `rwa.d` script shown next traces file I/O at the system call layer and will always match the correct process name. It also shows that you can create D scripts using command-line arguments, reducing the need to do edits to measure specific system calls for specific processes. The script takes two arguments: the system call name and the process name. It assumes that the system call specified has a file descriptor as the first argument (`arg0`); it's up to you to choose system calls where that is the case. It is a simple matter to remove that requirement, changing the aggregation key and header also, to make the script more generic.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
```

continues

```
4
5  dtrace:::BEGIN { trace("Tracing... Output after 10 seconds, or Ctrl-C\n"); }
6
7  syscall::$1:entry
8  /execname == $$2/
9  {
10         self->fd = arg0;
11         self->st = timestamp;
12 }
13 syscall::$1:return
14 /self->st/
15 {
16         @iot[pid, probefunc, fds[self->fd].fi_pathname] = sum(timestamp - self->st);
17         self->fd = 0;
18         self->st = 0;
19 }
20 tick-10sec
21 {
22         normalize(@iot, 1000);
23         printf("%-8s %-8s %-32s %-16s\n", "PID", "SYSCALL", "PATHNAME", "TIME(us)");
24         printa("%-8d %-8s %-32s %-@16d\n", @iot);
25 }
```

Script rwa.d

Invoking this script requires passing the name of the system call to be measured as the first command-line argument (`$1`, line 7) and the process name used in the entry probe predicate (line 8). You would typically run this script after having done systemwide system call profiling and determining which processes are generating I/O. Here's a sample run:

```
solaris# ./rwa.d write java
Tracing... Output after 10 seconds, or Ctrl-C
PID      SYSCALL  PATHNAME                         TIME(us)
20710    write    /export/zones/...                43
21414    write    /export/zones/...                97
21413    write    /export/zones/...                131
2366     write    <unknown>                        2564
21407    write    /export/zones/........./networks 3547
21407    write    /export/zones/....g/drv/tnf.conf 32532
21407    write    /export/zones/....00871B2761d0s1 36564
21407    write    /export/zones/..../drv/ptsl.conf 50821
21407    write    /export/zones/....t/etc/pam.conf 51627
21407    write    /export/zones/....ig/drv/mm.conf 64201
21407    write    /export/zones/..../drv/ohci.conf 69296
21407    write    /export/zones/....nfo_shmmax.out 71929
21407    write    /export/zones/....v/ramdisk.conf 139327
21407    write    /export/zones/....el/drv/wc.conf 142931
21407    write    /export/zones/....l/drv/ptc.conf 183881
21407    write    /export/zones/....rv/pseudo.conf 187028
21407    write    /export/zones/....sks/dev-lL.err 202320
21407    write    /export/zones/..../ls-ld_tmp.out 217525
[...]
21407    write    /export/zones/....nfo_semvmx.out 1018789
21407    write    /export/zones/....fig/ipcs-a.out 1037886
21407    write    /export/zones/....v/sbusmem.conf 1190904
^C
```

The path names in this example have been truncated due to their length. We measured the time of `write(2)` system calls for all processes named `java`. Note the use of the sum aggregating function (line 16); the `TIME` values produced represent the total time spent writing to a particular file over the ten-second sampling period (line 20). It's easy to change the aggregating function from `sum` to, for example, `avg` to get average times or `quantize` to get a distribution.

See Chapters 4 and 5 on disk I/O and file systems to dig deeper into these areas with DTrace.

Network I/O

As with disk I/O, network I/O begins with system calls from applications. In addition to the system calls listed in the "Disk I/O" section, applications performing network I/O may use `getmsg(2)` and `putmsg(2)`, as well as any number of section `3SOCKET` interfaces, many of which are implemented as system calls (for example, `recv(2)`, `recvfrom(2)`, `send(2)`, `sendto(2)`, and so on). For the most part, applications that perform network I/O use standard socket interfaces, which, on Solaris systems, are implemented via the `sockfs` file system. This makes connecting network I/O activity to processes and threads a snap. Here's a script that builds on the `sockfs` example in the "One-Liners" section:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt:sockfs::entry
{
        @[execname, probefunc] = count();
}
END
{
        printf("%-16s %-24s %-8s\n", "EXEC", "SOCKFS FUNC", "COUNT");
        printa("%-16s %-24s %-@8d\n", @);
}
```

Script sock.d

Here, and in the script that follows (`sock_j.d`), we leverage the existence of the `sockfs` layer in Solaris to connect network activity to the calling processes. For Mac OS X and FreeBSD, which do not implement sockets with `sockfs`, the fbt provider can be used with a blank field for the `probemod` and a wildcard character (`*`) in the `probefunc` field, using the entry probe name:

```
[root@freebsd /sys]# dtrace -n 'fbt::*socket*:entry
    { @[execname,probefunc] = count(); }'
dtrace: description 'fbt::*socket*:entry ' matched 41 probes
^C

  sendmail       mac_socket_check_poll                                     1
  sshd           mac_socket_check_receive                                  1
  sshd           mac_socket_check_send                                     1
  syslogd        mac_socket_check_poll                                     2
  sshd           mac_socket_check_poll                                     5
macosx# dtrace -n 'fbt::*socket*:entry { @[execname,probefunc] = count(); }'
dtrace: description 'fbt::*socket*:entry ' matched 31 probes
^C
[...]
  Safari         socket_lock                                             512
  VBoxSVC        socket_unlock                                          1855
  VirtualBoxVM   socket_unlock                                          2091
  VBoxSVC        socket_lock                                            2395
  VirtualBoxVM   socket_lock                                            2670
  VBoxXPCOMIPCD  socket_unlock                                          5152
  VBoxXPCOMIPCD  socket_lock                                            5824
```

These one-liners can be improved; some socket functions may contain abbreviated versions of `socket` (such as `sock` or `so`) and so will not be matched by the previous probe name.

Here's a sample run of the `sock.d` script on Solaris:

```
solaris# ./sock.d
^C
EXEC             SOCKFS FUNC              COUNT
gnome-panel      socktpi_ioctl            1
m2loader         getsonode                1
. . .
oracle           so_update_attrs          4855
java             so_lock_read_intr        4984
java             so_unlock_read           4984
java             socktpi_read             4984
java             sotpi_recvmsg            4984
java             so_update_attrs          7525
```

The output from the `sock.d` script gives us a good view into which processes are hitting the socket layer of the kernel and thus generating network I/O, and from the names of the `sockfs` functions, we can often infer the type of operation.

Here's the next drill-down:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt:sockfs::entry
/execname == "java"/
```

```
7  {
8          @[probefunc] = count();
9          self->st[stackdepth] = timestamp;
10
11 }
12 fbt:sockfs::return
13 /self->st[stackdepth]/
14 {
15         @sockfs_times[pid, probefunc] = sum(timestamp - self->st[stackdepth]);
16         self->st[stackdepth] = 0;
17 }
18 tick-1sec
19 {
20         normalize(@sockfs_times, 1000);
21         printf("%-8s %-24s %-16s\n", "PID", "SOCKFS FUNC", "TIME(ms)");
22         printa("%-8d %-24s %-@16d\n", @sockfs_times);
23
24         printf("\nSOCKFS CALLS PER SECOND:\n");
25         printa(@);
26
27         trunc(@); trunc(@sockfs_times);
28         printf("\n\n");
29 }
```

Script sock_j.d

The sock_j.d script includes some changes to drill down on just java processes, enabling us to take execname out as an aggregation key and replace it with pid. We capture a time stamp in the entry probe and do the math in the return probe to track the total time spent in the various socket functions. The time stamp is saved in self->st[stackdepth], which associates not only the time with the current thread (self->st) but also the time with the current level of the stack by using the stackdepth built-in. This is because the entry probe may fire multiple times as sockfs subfunctions are called, before the return probe is fired. Using stackdepth as a key associates each entry with the correct return, despite the subfunction calls.

In the tick probe, we convert the times from nanoseconds to microseconds using the normalize() function (line 20). We generate formatted output every second, which is an important component of using the time stamp function in DTrace. That is, when measuring the amount of time spent in different areas of the code, it's important to capture that information at predetermined intervals. This makes it much easier to determine whether the amount of time spent in a given function is relatively high or insignificant.

```
solaris# ./sock_j.d
PID      SOCKFS FUNC              TIME(ms)
1819     getsonode                44
1819     so_lock_read_intr        89
1819     so_unlock_read           99
```

continues

```
 1819     so_update_attrs          137
 1819     sostream_direct          212
 1819     socktpi_ioctl            264
 1819     sotpi_sendmsg            328
 1819     sendit                   639
 1819     send                     723
 21281    getsonode                6541
 21674    getsonode                7407
 21674    so_unlock_read           11478
 21674    so_lock_read_intr        11807
 21281    so_lock_read_intr        13271
 21281    so_unlock_read           13835
 21674    so_update_attrs          17931
 21281    so_update_attrs          20536
 21281    sostream_direct          21309
 21674    sostream_direct          21580
 21674    sotpi_sendmsg            35054
 21281    sotpi_sendmsg            36592
 21674    sendit                   74725
 21281    sendit                   78475
 21674    send                     85668
 21281    send                     90297
 1819     sotpi_recvmsg            114010
 1819     socktpi_read             114382
 21674    sotpi_recvmsg            340786
 21674    socktpi_read             387945
 21281    sotpi_recvmsg            581747
 21281    socktpi_read             637785

 SOCKFS CALLS PER SECOND:

   socktpi_ioctl                                               60
   getsonode                                                 4508
   send                                                      4508
   sendit                                                    4508
   sostream_direct                                           4508
   sotpi_sendmsg                                             4508
   so_lock_read_intr                                         9016
   so_unlock_read                                            9016
   socktpi_read                                              9016
   sotpi_recvmsg                                             9016
   so_update_attrs                                          13524
```

The sample output shows the time spent by specific `java` processes in network functions in the kernel. PID 21281 spent 638 milliseconds in `socktpi_read()`, 582 milliseconds in `sotpi_recvmsg()`, and so on. Since these functions include subfunction calls, the times overlap; the function call with the highest time is likely to be the highest in the stack and include the others. In this case, that would be `socktpi_read()`, which can be confirmed with some DTrace investigation of kernel stacks.

Given the one-second sampling interval, we can see that this particular process spent a significant percentage of time in socket I/O. Even though the kernel function `so_update_attrs()` was called the most frequently (13,524 times in the one-second interval), the time spent in that code was relatively small (17 milliseconds to 20 milliseconds). This illustrates the importance of measuring not just rates but time as well. Latency (time) matters most for application performance.

Examining the output of a particular D program can often lead to other questions. Looking at the sample shown earlier, it may be interesting to understand where the calls to the `so_update_attrs()` kernel socket module are originating. We can get that answer with a simple DTrace command line to grab a kernel stack when that function is entered:

```
solaris# dtrace -n 'fbt:sockfs:so_update_attrs:entry
    /execname == "java"/ { @[stack()] = count(); }'
dtrace: description 'fbt:sockfs:so_update_attrs:entry ' matched 1 probe
^C
              sockfs`socktpi_read+0x32
              genunix`fop_read+0x31
              genunix`read+0x188
              genunix`read32+0xe
              unix`sys_syscall32+0x101
                3

              sockfs`socktpi_write+0x161
              genunix`fop_write+0x31
              genunix`write+0x287
              unix`sys_syscall+0x17b
              195

              sockfs`sendit+0x17d
              sockfs`send+0x6a
              unix`sys_syscall+0x17b
             1126

              sockfs`socktpi_read+0x32
              genunix`fop_read+0x31
              genunix`read+0x188
              unix`sys_syscall+0x17b
             2481
```

The command line enabled a probe at the entry point of the kernel function of interest, used a predicate since we were looking at `java` processes, and aggregated on kernel stack frames. We can see from the output that the `so_update_attrs()` kernel function gets called when the application code reads and writes sockets. It also confirms that `socktpi_read()` was the highest `sockfs` function in the stack.

Another way to determine which processes are generating network I/O is to use the DTrace `fds[]` array and track I/O system calls to the `sockfs` file system.

```
solaris# dtrace -n 'syscall::*read:entry,syscall::*write:entry
    /fds[arg0].fi_fs == "sockfs"/ { @[execname] = count(); }'
dtrace: description 'syscall::*read:entry,syscall::*write:entry ' matched 4 probes
^C
```

continues

```
  xscreensaver                                                   3
  ssh                                                            4
  sshd                                                           6
  clock-applet                                                   7
  sge_execd                                                     36
  tnslsnr                                                       48
  m2loader                                                     558
  sge_qmaster                                                  658
  gnome-terminal                                              1132
  oracle                                                      1202
  Xvnc                                                        1380
  java                                                        3096
  mysqld                                                      3617
```

Once again, a few tweaks to the command line, and we can drill down on a process of choice, measuring (for example) the requested number of bytes to read, per second:

```
solaris# dtrace -qn 'syscall::read:entry /execname == "mysqld"
    && fds[arg0].fi_fs == "sockfs"/ { @rd_bytes = sum(arg2); }
    tick-1sec { printa(@rd_bytes); trunc(@rd_bytes); }'

          8755050

          7537546

         50807026

            32211

         10020337

            21858
^C
```

Here we tracked the mysqld process, focusing on read bandwidth. With the per-second ranges showing a pretty wide spread, it may be more interesting to use `quantize()` for a distribution of read sizes by this process.

```
solaris# dtrace -qn 'syscall::read:entry / execname == "mysqld"
    && fds[arg0].fi_fs == "sockfs" / { @rd_bytes = quantize(arg2); }'

^C


           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |                                         3
               2 |                                         0
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@                10160
               8 |                                         16
              16 |                                         76
              32 |                                         67
```

```
              64 |                                         39
             128 |@@@@@                                    2188
             256 |@@@                                      1225
             512 |@@                                       679
            1024 |@                                        413
            2048 |@                                        204
            4096 |                                         74
            8192 |                                         52
           16384 |                                         48
           32768 |                                         59
           65536 |                                         75
          131072 |                                         90
          262144 |                                         133
          524288 |                                         177
         1048576 |                                         104
         2097152 |                                         64
         4194304 |                                         36
         8388608 |                                         0
```

We removed the `tick` probe and chose to sample for a few seconds before hitting Ctrl-C. This gives us a better view of the distribution of the size of network reads for this process. We see that most are very small (4 bytes to 8 bytes), with a large percentage in the 128-byte to 1024-byte range.

As was the case with disk I/O, determining which processes are generating network I/O can be a good place to start, but it's also important to understand rates, throughput, and latency. We can use scripts like we used them in the "Disk I/O" section to obtain this information.

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  syscall::*read:entry,
6  syscall::*write:entry
7  /fds[arg0].fi_fs == "sockfs"/
8  {
9          @ior[probefunc] = count();
10         @net_bytes[probefunc] = sum(arg2);
11 }
12 tick-1sec
13 {
14         printf("%-8s %-16s %-16s\n", "FUNC", "OPS PER SEC", "BYTES PER SEC");
15         printa("%-8s %-@16d %-@16d\n", @ior, @net_bytes);
16         trunc(@ior); trunc(@net_bytes);
17         printf("\n");
18 }
```

Script net.d

Note the `net.d` script captures data in two aggregations (lines 9 and 10) to measure the rate of the calls and the amount of data requested to be read or written by the application. The printa() statement (line 15) leverages DTrace's ability to display multiple aggregations in one printa() call.

```
solaris# ./net.d
FUNC     OPS PER SEC      BYTES PER SEC
write    5009             675705
read     14931            12102460

FUNC     OPS PER SEC      BYTES PER SEC
write    5123             1126565
read     15698            14811819

FUNC     OPS PER SEC      BYTES PER SEC
write    5658             1299127
read     16977            16165513

FUNC     OPS PER SEC      BYTES PER SEC
write    4782             673129
read     14179            11532204

FUNC     OPS PER SEC      BYTES PER SEC
write    3690             2442080
read     10941            30772527
. . .
```

This system view of network activity indicates that we're doing substantially more reads than writes (by a factor of about three to one) and generating commensurately more read throughput than write throughput. However, this may not represent the entire picture for network traffic on the system. If applications are using other APIs, such as getmsg(2) and putmsg(2) to read and write network data, we would need to add those interfaces to the script or create a new script to acquire the same information. Because getmsg(2) and putmsg(2) have a very different argument list than read(2) and write(2) variants, obtaining the same information requires changes to the probe actions. This is true for send(2) and recv(2) as well, which are also used by applications doing network I/O. Chapter 6, Network Lower-Level Protocols, has several examples of scripts using these interfaces.

Reference Chapter 6 and Chapter 7, Application Protocols, for digging deeper into networking with DTrace.

Summary

Any performance or capacity analysis begins with a concise description of the performance problem in terms of something that can be measured. Taking a look at the overall system is an essential starting point, because it provides an understanding of the system and workload profile that may make your path to the root cause much shorter and/or may uncover other issues that might not otherwise have been visible. This chapter was intended to provide a starting point for your work with DTrace. Use the remaining chapters in this book for more detailed observability and to drill down into specific areas.

This page intentionally left blank

4

Disk I/O

Disk I/O is one of the most common causes of poor system and application performance. On the latency scale, CPU speeds are measured in gigahertz, memory access takes tens to hundreds of nanoseconds, and network packets make round-trips in microseconds. Disk reads and writes are at the far edge of this time scale, with a typical disk I/O taking several milliseconds. Thus, when we profile application latency, we must measure disk I/O and determine all aspects of a workload's disk I/O attributes (which files are being read and written, I/O sizes, I/O latency, throughput, and so on) in order to understand application behavior and performance.

DTrace can observe not only details of each disk I/O event but also the inner workings of disk device drivers, storage controller drivers, file systems, system calls, and the application that is requesting I/O. You can use it to answer questions such as the following.

- What is the pattern of disk access, address, and size?
- Which files in which file systems are being read or written?
- What are the highest latencies the disks are returning?
- Which processes/threads are causing disk I/O, and why?

As an example, `iosnoop` is a DTrace-based tool to trace disk I/O that ships with Mac OS X and OpenSolaris. It prints details of disk I/O events as they occur, including the I/O size and process name. The following shows the StarOffice application being launched, which causes thousands of disk I/O events:

```
# iosnoop
  UID   PID D    BLOCK   SIZE       COMM PATHNAME
    0  1337 R  5596978   7168       bash /usr/opt/staroffice8/program/soffice
    0  1341 R    52496   6144    soffice /usr/bin/basename
    0  1342 R    58304   8192    soffice /usr/bin/sed
    0  1342 R    58320   8192    soffice /usr/bin/sed
    0  1344 R    53776   6144    soffice /usr/bin/dirname
    0  1349 R  5646208   8192    soffice /usr/opt/staroffice8/program/javaldx
    0  1349 R  5643226   3072    soffice /usr/opt/staroffice8/program/javaldx
    0  1349 R  5512544   8192    javaldx <none>
    0  1349 R  5636672   8192    javaldx /usr/opt/staroffice8/program/libuno_sal.so.3
[...truncated...]
    0  1356 R 12094968   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094952   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12095000   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094840   8192       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094992   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094216   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12093960   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094016   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094672   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
    0  1356 R 12094736   4096       java /usr/j2se/jre/lib/i386/client/libjvm.so
[...truncated...]
```

The `iosnoop` program shows the bash shell referencing the `soffice` binary, `soffice` reading several command binaries, and a Java virtual machine starting. While the Java process loads `libjvm.so`, the I/O size is often 4KB, and the block location is somewhat random. Because this is accessing a rotating disk, small, random I/O is expected to perform poorly. This is an example of high-level information identifying a potential issue;[1] DTrace can dig much deeper as required.

The `iosnoop` program is explained in detail later in this chapter.

Capabilities

As we explore DTrace's capabilities for examining disk I/O, we will reference the functional I/O software stack shown in Figure 4-1, based on the Solaris I/O subsystem.

DTrace is capable of tracing every software component of the I/O stack, with the exception of physical disk drive internals.[2] This being the case, one of the hardest things for beginners is to decide what to do with it. DTrace can answer any question, but what question should we ask?

1. One solution to this type of issue is to use an application to prefetch the library into DRAM cache using sequential large I/O, before the application is launched.
2. An example is the operation of the onboard Disk Data Controller, since it is a dedicated silicon chip inside the actual hard drive; however, DTrace can examine all the requests and responses to disk at multiple layers and can infer internal disk behavior.

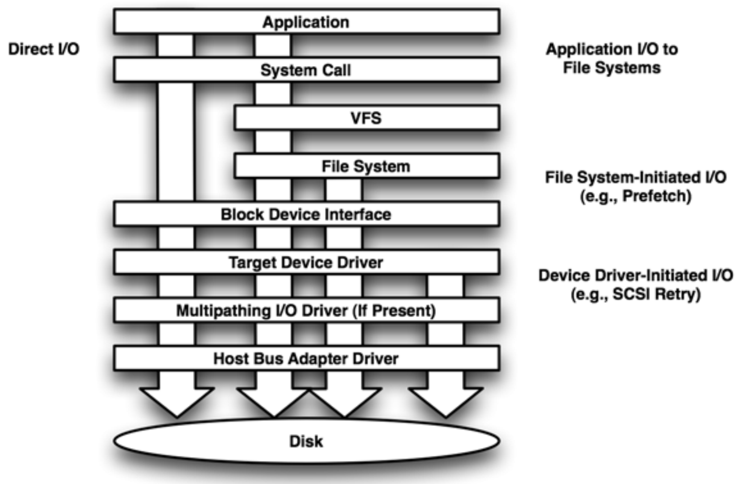


Figure 4-1 Functional diagram of Solaris I/O stack

Figure 4-2 shows an abstract I/O module. At each of the numbered items, we can ask questions such as the following.

1. What are the requests? What type, how many, and what I/O size?
2. What was rejected, and why?
3. How long did the request processing take (on-CPU)?
4. If a queue exists, what is the average queue length and wait time?
5. What made it to the next level? How does it compare to 1, and was the ordering the same?
6. How long did the I/O take to return, and how many are in-flight?
7. What's the I/O latency (includes queue and service time)? Decompose latency by type.
8. What was the error latency?
9. How long did response processing take (on-CPU)?
10. What completed, and how does that compare to 5 and 1. Was the ordering the same?
11. What errors occurred, and why?
12. What I/Os were retried?
13. Are timeouts occurring?
14. How long were the timeout errors?

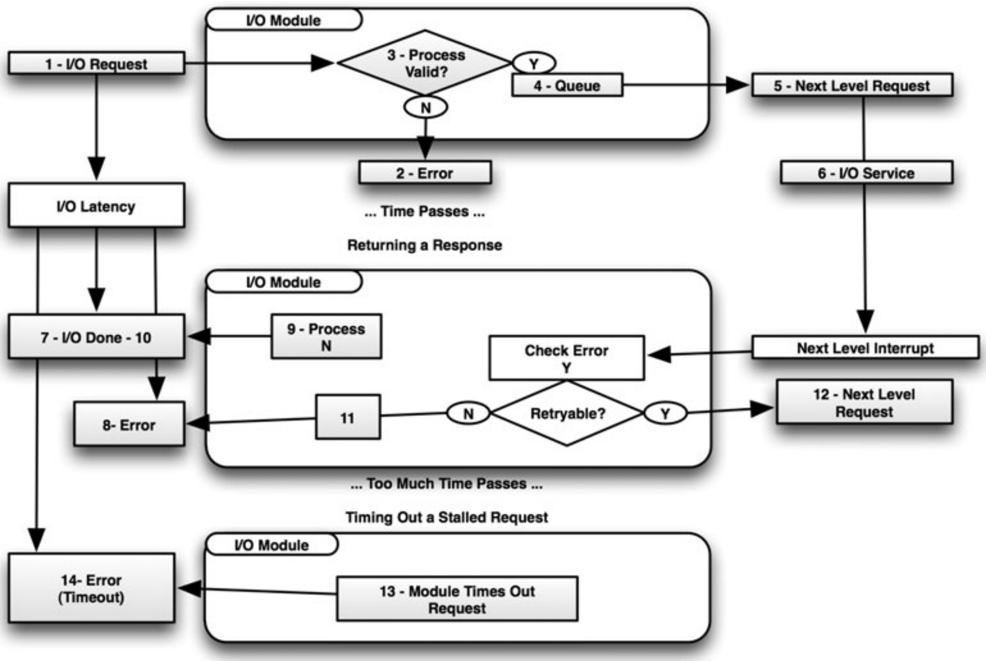


Figure 4-2 Generic I/O module internals

Real system I/O components (hardware and software) will process I/O more or less as shown in Figure 4-2 (block device driver, SCSI, SATA, and so on). Some components may not have queues, or they may not retry or time out requests. If you can find a similar internal diagram for the I/O module that interests you, then you should be able to identify good targets to DTrace in a similar fashion.

Disk I/O Strategy

To get started using DTrace to examine disk I/O, follow these steps (the target of each step is in bold):

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.
2. In addition to those DTrace tools, familiarize yourself with existing **disk statistic tools**, such as `iostat(1M)`. The metrics that these generate can be treated as starting points for customization with DTrace; the `rwtime.d` script is an example.

3. Locate or write tools to generate **known workloads** of disk I/O, such as running the `dd(1)` command to read from a raw disk device (under `/dev/rdsk` on Solaris). When writing your own disk I/O tools, it is *extremely* helpful to have known workloads to check them against.
4. Customize and write your own one-liners and scripts using the io provider, referring to the **io provider** documentation in the "Providers" section.
5. To dig deeper than the io provider allows, familiarize yourself with how the kernel and user-land processes call I/O by examining **stack backtraces** (see the "One-Liners" section). Also refer to functional diagrams of the I/O subsystem, such as those shown earlier and those in published kernel texts such as *Solaris Internals* (McDougall and Mauro, 2006).
6. Examine kernel internals for file systems and device drivers by using the **fbt provider** and referring to kernel source code (if available). Write scripts to examine higher-level details first (I/O counts), and then drill down deeper into areas of interest.

Checklist

Table 4-1 suggests different types of issues that can be examined using DTrace. This can also serve as a checklist to ensure that you consider all obvious types of issues.

Table 4-1 Disk I/O Checklist

Issue	Description
Volume	System and applications may be performing a high volume of disk I/O, which can be avoided by changing their behavior, for example, by using or tuning a higher-level cache. DTrace can be used to examine disk I/O by process, filename, size, and stack trace, to identify what is using the disks and by how much.
Service Time	For rotating magnetic disks, I/O latency for random disk accesses can be many milliseconds, throttling application throughput. Flash-based solid-state disks have submillisecond I/O latency. However, some types of solid-state disks will still exhibit high latencies for writes. Use DTrace to examine disk I/O latency.
Queueing	High I/O latency can also be caused by I/O queueing, rather than the disk time to service that I/O. Use DTrace to examine pending I/O, including cases where bursts of I/O are sent to the disk, causing I/O to wait on the queue.

continues

Table 4-1 Disk I/O Checklist (*Continued*)

Issue	Description
Errors	Disks can malfunction and can lead to latencies on the order of seconds while the disk retries the operation (this may not be reported by the standard operating system tools!). This may cause an application to experience slow I/O for no clear reason. DTrace can be used to examine errors, retries, and timeouts from all layers of the I/O subsystem.
Configuration	Disks often support features such as read and write caching and command queueing, which can greatly affect performance when enabled. Other configurable options may include multipathing to the disks. DTrace can be used to check that such options are enabled and working as expected.

To identify performance issues, focus on the time spent waiting for disk I/O to complete: the I/O latency for the entire operation (queueing + service). I/O operations per second (IOPS), throughput, I/O size, and disk address all shed light on the nature of the I/O. However, *latency* identifies whether this I/O is causing a problem and can quantify the extent of it.

For example, if an application is performing transactions that include disk I/O, the ideal DTrace script would show disk I/O time as a percentage ratio of transaction time. Other components of that transaction time might include CPU time, network I/O time, lock contention, and thread dispatcher queue latency, all of which can also be measured using DTrace.

Providers

Table 4-2 shows providers that you can use to trace disk I/O.

Table 4-2 Providers for Disk I/O

Provider	Description
io	Stable I/O provider. Traces disk I/O (and other back-end storage devices).
sdt	Statically Defined Tracing provider. Includes deliberately placed DTrace probes of interest, but the interface is considered unstable and may change.
fbt	Function Boundary Tracing provider. Used to examine internals of the I/O subsystem and drivers in detail. This has an unstable interface and will change between releases of the operating system and drivers, meaning that scripts based on fbt may need to be slightly rewritten for each such update. See the "fbt Provider" section in Chapter 12, Kernel.

io Provider

The io provider traces I/O events in the kernel. Its behavior varies slightly between operating systems:

Solaris: Traces disk I/O and NFS client back-end I/O

Mac OS X: Traces disk I/O

FreeBSD: Not yet available (see the "fbt Provider" section)

For simplicity, many of the one-liners and scripts in this chapter describe the io provider as tracing "disk I/O"; be aware that on Solaris it can also trace NFS client back-end I/O (see the "Matching Disk I/O Only" section that follows for how to avoid this).

The io provider design sets an excellent example for DTrace providers in general. It presents data from complex kernel structures in a stable, intuitive, and user-friendly way, encouraging the analysis of disk I/O events from kernel context. For simplicity, the probes have been kept to the minimum: `start`, `done`, `wait-start`, and `wait-done` (see Table 4-3). The arguments are also kept simple and easy to follow, presenting details about the I/O, the device, and the file system. For reference, the io provider specification has been reproduced from the DTrace Guide[3] in the following pages to illustrate these points.

Table 4-3 io Probes

Probe	Description
`start`	Fires when an I/O request is about to be made to a peripheral device or to an NFS server. The `bufinfo_t` corresponding to the I/O request is pointed to by `args[0]`. The `devinfo_t` of the device to which the I/O is being issued is pointed to by `args[1]`. The `fileinfo_t` of the file corresponding to the I/O request is pointed to by `args[2]`. Note that file information availability depends on the file system making the I/O request. See the information about `fileinfo_t` for more information.
`done`	Fires after an I/O request has been fulfilled. The `bufinfo_t` corresponding to the I/O request is pointed to by `args[0]`. The `done` probe fires after the I/O completes but before completion processing has been performed on the buffer. As a result, `B_DONE` is *not* set in `b_flags` when the `done` probe fires. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file corresponding to the I/O request is pointed to by `args[2]`.

continues

3. You can currently find this at *http://wikis.sun.com/display/DTrace/Documentation*.

Table 4-3 io Probes (*Continued*)

Probe	Description
`wait-start`	Fires after an I/O request has been fulfilled. The `bufinfo_t` corresponding to the I/O request is pointed to by `args[0]`. The `done` probe fires after the I/O completes but before completion processing has been performed on the buffer. As a result, `B_DONE` is *not* set in `b_flags` when the `done` probe fires. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file corresponding to the I/O request is pointed to by `args[2]`.
`wait-done`	Fires on the completion of an I/O request. The `bufinfo_t` corresponding to the I/O request for which the thread will wait is pointed to by `args[0]`. The `devinfo_t` of the device to which the I/O was issued is pointed to by `args[1]`. The `fileinfo_t` of the file corresponding to the I/O request is pointed to by `args[2]`. The `wait-done` probe fires only after the `wait-start` probe has fired in the same thread.

bufinfo_t

The `bufinfo_t` structure is the abstraction describing an I/O request. The buffer corresponding to an I/O request is pointed to by `args[0]` in the `start`, `done`, `wait-start`, and `wait-done` probes. The `bufinfo_t` structure definition is as follows:

```
typedef struct bufinfo {
        int b_flags;                    /* buffer status flags */
        size_t b_bcount;                /* number of bytes */
        caddr_t b_addr;                 /* buffer address */
        uint64_t b_lblkno;              /* block # on device */
        uint64_t b_blkno;               /* expanded block # on device */
        size_t b_resid;                 /* # of bytes not transferred */
        size_t b_bufsize;               /* size of allocated buffer */
        caddr_t b_iodone;               /* I/O completion routine */
        int b_error;                    /* expanded error field */
        dev_t b_edev;                   /* extended device */
 } bufinfo_t;
```

The structure members are as follows.

The `b_flags` member indicates the state of the I/O buffer and consists of a bitwise-OR of different state values. The valid state values are shown in Table 4-4.

The `b_bcount` field is the number of bytes to be transferred as part of the I/O request.

The `b_addr` field is the virtual address of the I/O request, unless `B_PAGEIO` is set. The address is a kernel virtual address unless `B_PHYS` is set, in which case it is a user virtual address. If B_PAGEIO is set, the b_addr field contains kernel private data. Exactly one of B_PHYS and B_PAGEIO can be set, or neither will be set.

The b_lblkno field identifies which logical block on the device is to be accessed. The mapping from a logical block to a physical block (such as the cylinder, track, and so on) is defined by the device.

The b_resid field is set to the number of bytes not transferred because of an error.

The b_bufsize field contains the size of the allocated buffer.

The b_iodone field identifies a specific routine in the kernel that is called when the I/O is complete.

The b_error field may hold an error code returned from the driver in the event of an I/O error. b_error is set in conjunction with the B_ERROR bit set in the b_flags member.

The b_edev field contains the major and minor device numbers of the device accessed. Consumers may use the D subroutines getmajor and getminor to extract the major and minor device numbers from the b_edev field.

Table 4-4 b_flags Values

Provider	Description
B_DONE	Indicates that the data transfer has completed.
B_ERROR	Indicates an I/O transfer error. It is set in conjunction with the b_error field. This flag may exist only on Solaris; for other operating systems, check for a nonzero value of b_error to identify errors.
B_PAGEIO	Indicates that the buffer is being used in a paged I/O request. See the description of the b_addr field for more information.
B_PHYS	Indicates that the buffer is being used for physical (direct) I/O to a user data area.
B_READ	Indicates that data is to be read from the peripheral device into main memory.
B_WRITE	Indicates that the data is to be transferred from main memory to the peripheral device.
B_ASYNC	The I/O request is asynchronous and will not be waited for. The wait-start and wait-done probes don't fire for asynchronous I/O requests. Note that some I/Os directed to be asynchronous might not have B_ASYNC set: the asynchronous I/O subsystem might implement the asynchronous request by having a separate worker thread perform a synchronous I/O operation.

devinfo_t

The devinfo_t structure provides information about a device. The devinfo_t structure corresponding to the destination device of an I/O is pointed to by args[1] in the start, done, wait-start, and wait-done probes. The members of devinfo_t are as follows:

```
typedef struct devinfo {
        int dev_major;              /* major number */
        int dev_minor;              /* minor number */
        int dev_instance;           /* instance number */
        string dev_name;            /* name of device */
        string dev_statname;        /* name of device + instance/minor */
        string dev_pathname;        /* pathname of device */
} devinfo_t;
```

The dev_major field is the major number of the device.

The dev_minor field is the minor number of the device.

The dev_instance field is the instance number of the device. The instance of a device is different from the minor number. The minor number is an abstraction managed by the device driver. The instance number is a property of the device node.

The dev_name field is the name of the device driver that manages the device, if available.

The dev_statname field is the name of the device as reported by system administration tools such as iostat(1M), if available.

The dev_pathname field is the full path of the device. The path specified by dev_pathname includes components expressing the device node, the instance number, and the minor node. However, all three of these elements aren't necessarily expressed in the statistics name. For some devices, the statistics name consists of the device name and the instance number. For other devices, the name consists of the device name and the number of the minor node. As a result, two devices that have the same dev_statname may differ in dev_pathname.

On Mac OS X, dev_name, dev_statname, and dev_pathname may not be available and return the string ??. In this case, devices may still be identified by their major and minor numbers.

fileinfo_t

The fileinfo_t structure provides information about a file. The file to which an I/O corresponds is pointed to by args[2] in the start, done, wait-start, and wait-done probes. The presence of file information is contingent upon the file system providing this information when dispatching I/O requests. Some file systems, especially third-party file systems, might not provide this information. Also, I/O requests might emanate from a file system for which no file information exists. For example, any I/O to file system metadata will not be associated with any one file. Finally, some highly optimized file systems might aggregate I/O from disjoint files into a single I/O request. In this case, the file system might provide the file information either for the file that represents the majority of the I/O or for the file that represents *some* of the I/O; or, the file system might provide no file information at all.

The definition of the fileinfo_t structure is as follows:

```
typedef struct fileinfo {
        string fi_name;         /* name (basename of fi_pathname) */
        string fi_dirname;      /* directory (dirname of fi_pathname) */
        string fi_pathname;     /* full pathname */
        offset_t fi_offset;     /* offset within file */
        string fi_fs;           /* file system */
        string fi_mount;        /* mount point of file system */
} fileinfo_t;
```

The fi_name field contains the name of the file but does not include any directory components. If no file information is associated with an I/O, the fi_name field will be set to the string <none>. In some cases, the path name associated with a file might be unknown. In this case, the fi_name field will be set to the string <unknown>. On Mac OS X, this string may also contain a reason in parentheses, for example, <unknown (NULL v_name)>.

The fi_dirname field contains *only* the directory component of the filename. As with fi_name, this string may be set to <none> if no file information is present or to <unknown> if the path name associated with the file is not known.

The fi_pathname field contains the full path name to the file. As with fi_name, this string may be set to <none> if no file information is present or to <unknown> if the path name associated with the file is not known.

The fi_offset field contains the offset within the file or contains -1 if file information is not present or if the offset is otherwise unspecified by the file system.

Command-Line Hints

At the command line, you can use the -v switch with dtrace(1M) as a reminder of which arguments belong to which io provider probes:

```
solaris# dtrace -lvn io:::start
[...]
24463         io           genunix                      bdev_strategy start
[...]
        Argument Types
                args[0]: bufinfo_t *
                args[1]: devinfo_t *
                args[2]: fileinfo_t *
```

And, as a reminder of the members of these arguments, you can read the translator for the io provider, which is usually in `/usr/lib/dtrace/io.d`:

```
solaris# more /usr/lib/dtrace/io.d
[...]
typedef struct bufinfo {
        int b_flags;                     /* buffer status */
        size_t b_bcount;                 /* number of bytes */
        caddr_t b_addr;                  /* buffer address */
[...etc...]
```

The translator file provides the stable argument interface for the io provider, from raw kernel data. Since the language is D, it can be easily read for interesting insight into how this information is retrieved from the kernel. For example, the mount-point path `fi_mount` is translated differently between Solaris and Mac OS X.

Here it is on Solaris:

```
fi_mount = B->b_file == NULL ? "<none>" :
            B->b_file->v_vfsp->vfs_vnodecovered == NULL ? "/" :
            B->b_file->v_vfsp->vfs_vnodecovered->v_path == NULL ? "<unknown>" :
            cleanpath(B->b_file->v_vfsp->vfs_vnodecovered->v_path);
```

Here it is on Mac OS X:

```
fi_mount = B->b_vp->v_mount->mnt_vnodecovered == NULL ? "/" :
    B->b_vp->v _mount->mnt_vnodecovered->v_name;
```

This translation code may change in future updates to the kernels, but the interface provided will remain the same, so scripts written using io probes will continue to work. This is how DTrace is able to maintain stable providers when underlying implementation details change.

Matching Disk I/O Only

The io provider on Solaris and OpenSolaris also traces back-end NFS I/O, which can be seen when listing probes:

```
solaris# dtrace -ln io:::start
   ID   PROVIDER            MODULE                          FUNCTION NAME
  755         io           genunix                   default_physio start
  756         io           genunix                    bdev_strategy start
  757         io           genunix                          aphysio start
 2028         io               nfs                         nfs4_bio start
 2029         io               nfs                         nfs3_bio start
 2030         io               nfs                          nfs_bio start
```

Note the `io:::start` probes in the `nfs` kernel module. There may be times when you want to examine disk I/O on a client that is also performing NFS I/O and want to filter out the NFS I/O events. The probe description `io:genunix::start` would avoid matching `nfs` probes and only match the disk I/O probes; however, it also includes an unstable component—the module name—in what is otherwise a stable probe description. Module names are dynamically built based on the probe location in the source and are not part of the stable provider interface. Future versions of Solaris could move the disk I/O functions from `genunix` into another kernel module, or they could rename the `genunix` module entirely—either of which would cause D scripts based on `io:genunix::start` to stop working.

Instead of the probe description (which does work[4]), disk I/O can be matched exclusively by using the `io:::start` probe with the predicate: `/args[1]->dev_name != "nfs"/`.

fbt Provider

The fbt provider can be used to examine *all* the functions in the kernel I/O subsystem, function arguments, return codes, and elapsed time. Since it's tracing raw kernel code, any scripts are considered unstable and are likely to break between different kernel versions, which is why we list the fbt provider last in the "Strategy" section. See the "fbt Provider" section in Chapter 12 for more details, and the fbt provider chapter of the DTrace Guide[5] for the full reference.

To navigate this capability for disk I/O, kernel stack traces may be examined using DTrace to create a list of potential fbt probes and their relationships. Each line of the stack trace can be probed individually. Examining stack traces is also a quick way to became familiar with a complex body of code such as the I/O subsystem. We will demonstrate this for FreeBSD. Because the io provider is currently not available on FreeBSD, the fbt provider is the next best choice.

4. I've been guilty of using it, such as in many versions of the `iosnoop` tool.
5. You can find this currently at *http://wikis.sun.com/display/DTrace/fbt+Provider.*

To explore FreeBSD I/O, the `bsdtar(1)` command was used to create a disk read workload while the ATA disk driver strategy function was traced, aggregating on the process name and stack backtrace:

```
freebsd# dtrace -n 'fbt::ad_strategy:entry { @[execname, stack()] = count(); }'
dtrace: description 'fbt::ad_strategy:entry ' matched 1 probe
^C

  g_down
              kernel`g_disk_start+0x1a8
              kernel`g_io_schedule_down+0x269
              kernel`g_down_procbody+0x68
              kernel`fork_exit+0xca
              kernel`0xc0bc2040
             2863
```

The process name identified was `g_down`, with a short stack backtrace. This isn't showing the `bsdtar(1)` command creating the workload; rather, this has traced `GEOM(4)` (disk I/O transformation framework) performing the device I/O. To see the rest of the stack, the GEOM VFS strategy function can be traced to see who is requesting VFS-style I/O from GEOM.

```
freebsd# dtrace -n 'fbt::g_vfs_strategy:entry { @[execname, stack()] = count(); }'
dtrace: description 'fbt::g_vfs_strategy:entry ' matched 1 probe
^C
[...]
  bsdtar
              kernel`ffs_geom_strategy+0x14f
              kernel`ufs_strategy+0xd3
              kernel`VOP_STRATEGY_APV+0x8b
              kernel`bufstrategy+0x2e
              kernel`breadn+0xca
              kernel`bread+0x4c
              kernel`ffs_read+0x254
              kernel`VOP_READ_APV+0x7c
              kernel`vn_read+0x238
              kernel`dofileread+0x96
              kernel`kern_readv+0x58
              kernel`read+0x4f
              kernel`syscall+0x3e5
              kernel`0xc0bc2030
             2225
```

This has identified the correct process, `bsdtar(1)`, along with the stack trace down to the read system call. Any line from these stacks can be traced individually using the fbt provider so that details of the I/O can be examined. For more examples, see `bufstrategy()`. It is traced in the "One-Liners" section to see who is requesting disk I/O. The GEOM functions are traced in `geomiosnoop.d`.

One-Liners

The following one-liners should be used to begin your analysis of disk I/O.

io Provider

Trace disk I/O size by process ID:

```
dtrace -n 'io:::start { printf("%d %s %d", pid, execname, args[0]->b_bcount); }'
```

Show disk I/O size as distribution plots, by process name:

```
dtrace -n 'io:::start { @size[execname] = quantize(args[0]->b_bcount); }'
```

Identify user stacks when a process ID directly causes disk I/O:

```
dtrace -n 'io:::start /pid == $target/ { @[ustack()] = count(); }' -p PID
```

Identify user stacks when processes of a given name directly cause disk I/O, for example, `firefox-bin`:

```
dtrace -n 'io:::start /execname == "firefox-bin"/ { @[ustack()] = count(); }'
```

Identify kernel stacks calling disk I/O:

```
dtrace -n 'io:::start { @[stack()] = count(); }'
```

Trace errors along with disk and error number:

```
dtrace -n 'io:::done /args[0]->b_flags & B_ERROR/ { printf("%s err: %d"
 ,    args[1]->dev_statname, args[0]->b_error); }'
```

fbt Provider

The fbt provider instruments a particular operating system and version; these one-liners may therefore require modifications to match the software version you are running.

Here are the frequency count functions from disk driver (for example, `sd`):

```
dtrace -n 'fbt:sd::entry { @[probefunc] = count(); }'
dtrace -n 'fbt::sd_*:entry { @[probefunc] = count(); }'
```

Identify kernel stacks calling disk I/O (FreeBSD):

```
dtrace -n 'fbt::bufstrategy:entry { @[stack()] = count(); }'
```

Trace SCSI retries, showing `sd_lun` (Solaris):

```
dtrace -n 'fbt::sd_set_retry_bp:entry { printf("%x", arg0); }'
```

Count SCSI commands by SCSI code (Solaris):

```
dtrace -n 'fbt::scsi_transport:entry { @[*args[0]->pkt_cdbp] = count(); }'
```

Count SCSI packets by completion code (Solaris):

```
dtrace -n 'fbt::scsi_destroy_pkt:entry { @[args[0]->pkt_reason] = count(); }'
```

One-Liner Examples

Each of the one-liners is demonstrated in this section.

Disk I/O Size by Process ID

StarOffice was launched on Mac OS X while this one-liner was executing:

```
# dtrace -n 'io:::start { printf("%d %s %d", pid, execname, args[0]->b_bcount); }'
dtrace: description 'io:::start ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  18572               buf_strategy:start 189 Terminal 12288
  0  18572               buf_strategy:start 1688 soffice 73728
  0  18572               buf_strategy:start 1688 soffice 81920
  0  18572               buf_strategy:start 1688 soffice 4096
  0  18572               buf_strategy:start 1688 soffice 4096
  0  18572               buf_strategy:start 1688 soffice 724992
  0  18572               buf_strategy:start 1688 soffice 339968
  0  18572               buf_strategy:start 1688 soffice 16384
```

```
  0  18572              buf_strategy:start 1676 mdworker 12288
  0  18572              buf_strategy:start 1676 mdworker 4096
  1  18572              buf_strategy:start 1688 soffice 4096
  1  18572              buf_strategy:start 22 mds 4096
[...]
```

The previous example shows the `soffice` process calling some large physical disk I/Os, the largest more than 700KB. There are also several 4KB I/Os.

Disk I/O Size Aggregation

Here DTrace is used to determine the size of the disk I/O caused by the VirtualBox application running a virtual OS on Mac OS X:

```
# dtrace -n 'io:::start { @size[execname] = quantize(args[0]->b_bcount); }'
dtrace: description 'io:::start ' matched 1 probe
^C

  VirtualBoxVM
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@                  690
            8192 |@@@@@@@@@                                273
           16384 |@@@                                      76
           32768 |@@                                       53
           65536 |@                                        34
          131072 |@                                        28
          262144 |@                                        28
          524288 |@                                        23
         1048576 |                                         3
         2097152 |                                         0
```

Using the DTrace quantize aggregating function, we see that most of the physical disk I/O was between 4KB and 8KB while this script was tracing. We also see some large I/Os in the 256KB to 2MB range.

Identify User Stacks When a Process ID Causes Disk I/O

```
# dtrace -n 'io:::start /pid == $target/ { @[ustack()] = count(); }' -p 1721
dtrace: description 'io:::start ' matched 1 probe
^C

 libSystem.B.dylib`write+0xa
 VBoxDDU.dylib`vmdkWrite(void*, unsigned long long, void const*, ...
 VBoxDDU.dylib`vdWriteHelper(VBOXHDD*, VDIMAGE*, unsigned long long, ...
 VBoxDD.dylib`drvblockWrite(PDMIBLOCK*, unsigned long long,
    void const*, ...
 VBoxDD.dylib`ahciAsyncIOLoop(PDMDEVINS*, PDMTHREAD*)+0x3eb
 VBoxVMM.dylib`pdmR3ThreadMain(RTTHREADINT*, void*)+0xd5
 VBoxRT.dylib`rtThreadMain+0x40
 VBoxRT.dylib`rtThreadNativeMain(void*)+0x84
```

continues

```
   libSystem.B.dylib`_pthread_start+0x141
   libSystem.B.dylib`thread_start+0x22
               188

   libSystem.B.dylib`read+0xa
   VBoxDDU.dylib`vmdkRead(void*, unsigned long long, void*,
      unsigned long, ...
   VBoxDDU.dylib`vdReadHelper(VBOXHDD*, VDIMAGE*, unsigned long long, ...
   VBoxDD.dylib`ahciAsyncIOLoop(PDMDEVINS*, PDMTHREAD*)+0x4f9
   VBoxVMM.dylib`pdmR3ThreadMain(RTTHREADINT*, void*)+0xd5
   VBoxRT.dylib`rtThreadMain+0x40
   VBoxRT.dylib`rtThreadNativeMain(void*)+0x84
    libSystem.B.dylib`_pthread_start+0x141
    libSystem.B.dylib`thread_start+0x22
              2263
```

In this previous example, we can see the size of disk I/O that VirtualBox was sending. Now we'll frequency count the user stack traces when VirtualBoxVM (PID 1721) issues disk I/O in order to provide insight as to where in the VirtualBox code path the I/Os are initiated from.

Most of the disk I/O includes `vmdkRead` in the stack, which, at a guess, might be for a virtual machine disk read. Fetching stack traces is usually of most interest to the developers of the application, who have access to the source code, but it can still be useful for nondevelopers when trying to better understand the source of disk I/Os or when gathering additional information to send to developers for improving application code.

Identify Kernel Stacks Calling Disk I/O

While the previous user stack traces showed *why* the application caused disk I/O, by examining the kernel stack trace we can see *how* the disk I/O was called.

For Solaris and ZFS, note that the stack traces can become very long on ZFS; the `stackframes` tunable must be set to include the entire stack backtrace.

```
solaris# dtrace -x stackframes=64 -n 'io:::start { @[stack()] = count(); }'
dtrace: description 'io:::start ' matched 6 probes
^C
[...]
              genunix`ldi_strategy+0x59
              zfs`vdev_disk_io_start+0xd0
              zfs`zio_vdev_io_start+0x17d
              zfs`zio_execute+0x89
              zfs`zio_nowait+0x42
              zfs`vdev_mirror_io_start+0x148
              zfs`zio_vdev_io_start+0x17d
              zfs`zio_execute+0x89
              zfs`zio_nowait+0x42
              zfs`vdev_mirror_io_start+0x148
              zfs`zio_vdev_io_start+0x1ba
              zfs`zio_execute+0x89
              zfs`zio_nowait+0x42
              zfs`arc_read_nolock+0x81e
              zfs`arc_read+0x75
```

```
                zfs`dbuf_prefetch+0x134
                zfs`dmu_zfetch_fetch+0x8c
                zfs`dmu_zfetch_dofetch+0xb8
                zfs`dmu_zfetch_find+0x436
                zfs`dmu_zfetch+0xac
                zfs`dbuf_read+0x11c
                zfs`dmu_buf_hold_array_by_dnode+0x1c9
                zfs`dmu_buf_hold_array+0x6e
                zfs`dmu_read_uio+0x4d
                zfs`zfs_read+0x19a
                genunix`fop_read+0xa7
                nfssrv`rfs3_read+0x3a1
                nfssrv`common_dispatch+0x3a0
                nfssrv`rfs_dispatch+0x2d
                rpcmod`svc_getreq+0x19c
                rpcmod`svc_run+0x16e
                rpcmod`svc_do_run+0x81
                nfs`nfssys+0x765
                unix`sys_syscall32+0x101
                  4

                genunix`ldi_strategy+0x59
                zfs`vdev_disk_io_start+0xd0
                zfs`zio_vdev_io_start+0x17d
                zfs`zio_execute+0x89
                zfs`vdev_queue_io_done+0x92
                zfs`zio_vdev_io_done+0x62
                zfs`zio_execute+0x89
                genunix`taskq_thread+0x1b7
                unix`thread_start+0x8
               3702
```

The most frequent stack trace shows disk I/O being called from a `taskq_thread()`, which is part of the ZFS pipeline. Only a few of the disk I/Os originate directly from a system call, whose stack trace spans much of ZFS internals.

Here's the result on Mac OS X, HFS+:

```
macosx# dtrace -n 'io:::start { @[stack()] = count(); }'
dtrace: description 'io:::start ' matched 1 probe
^C
[...]
              mach_kernel`buf_strategy+0x60
              mach_kernel`hfs_vnop_strategy+0x34
              mach_kernel`VNOP_STRATEGY+0x2f
              mach_kernel`cluster_copy_upl_data+0xacf
              mach_kernel`cluster_copy_upl_data+0xec8
              mach_kernel`cluster_pageout+0x161a
              mach_kernel`cluster_push_ext+0xb1
              mach_kernel`cluster_push+0x28
              mach_kernel`GetLogicalBlockSize+0x631d
              mach_kernel`hfs_vnop_ioctl+0x305a
              mach_kernel`vnode_iterate+0x15c
              mach_kernel`hfs_mark_volume_inconsistent+0x27ed
              mach_kernel`VFS_SYNC+0x6f
              mach_kernel`mount_dropcrossref+0xf0
              mach_kernel`vfs_iterate+0xcc
              mach_kernel`sync+0x22
              mach_kernel`unix_syscall+0x23c
              mach_kernel`lo_unix_scall+0xea
               16
```

The stack trace spans from the system call (at the bottom) to the common disk I/O call via `buf_strategy()`. Stack frames in between show processing for VFS and then the HFS+ file system.

Identify Kernel Stacks Calling Disk I/O (FreeBSD)

Because FreeBSD does not yet have the io provider, the fbt provider is used instead to trace the kernel function, which requests buffer I/O, `bufstrategy()`:

```
freebsd# dtrace -n 'fbt::bufstrategy:entry { @[stack()] = count(); }'
dtrace: description 'fbt::bufstrategy:entry ' matched 1 probe
^C
[...]
              kernel`breadn+0xca
              kernel`bread+0x4c
              kernel`ffs_read+0x254
              kernel`VOP_READ_APV+0x7c
              kernel`vn_read+0x238
              kernel`dofileread+0x96
              kernel`kern_readv+0x58
              kernel`read+0x4f
              kernel`syscall+0x3e5
              kernel`0xc0bc2030
             1408
```

The most frequent stack traces show that the disk I/O was originating from read syscalls, `dofileread()`, `VOP_READ_APV()`, and `ffs_read()` (FreeBSD UFS).

Trace Errors Along with Disk and Error Number

To demonstrate tracing disk errors, we physically removed a disk during disk I/O:[6]

```
# dtrace -n 'io:::done /args[0]->b_flags & B_ERROR/
    { printf("%s err: %d", args[1]->dev_statname, args[0]->b_error); }'
dtrace: description 'io:::done ' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0  30197                     biodone:done sd128 err: 14
  0  30197                     biodone:done sd128 err: 14
  0  30197                     biodone:done sd128 err: 14
  0  30197                     biodone:done sd128 err: 14
  0  30197                     biodone:done sd128 err: 5
  1  30197                     biodone:done sd128 err: 14
  4  30197                     biodone:done sd128 err: 14
  4  30197                     biodone:done sd128 err: 5
  4  30197                     biodone:done sd128 err: 5
  3  30197                     biodone:done sd128 err: 5
```

6. This is not recommended. Nor is shouting at JBODs.

Errors 5 and 14 are from the Solaris `/usr/include/sys/errno.h` file:

```
#define EIO             5               /* I/O error */
#define EFAULT          14              /* Bad address */
```

These are the same on Mac OS X. There is also a DTrace translator for these error codes in `/usr/lib/dtrace/errno.h`, if you don't have an `errno.h` file to check.

Frequency Count Functions from Disk Driver (For Example, sd)

This shows tracing all the function calls from the SCSI disk driver on Solaris:

```
# dtrace -n 'fbt:sd::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:sd::entry ' matched 273 probes
^C

  sd_pm_idletimeout_handler                                          11
  ddi_xbuf_qstrategy                                                906
  sd_add_buf_to_waitq                                               906
  sd_core_iostart                                                   906
  sd_initpkt_for_buf                                                906
  sd_mapblockaddr_iostart                                           906
  sd_setup_rw_pkt                                                   906
  sd_xbuf_init                                                      906
  sd_xbuf_strategy                                                  906
  sdinfo                                                            906
  sdstrategy                                                        906
  xbuf_iostart                                                      906
  ddi_xbuf_done                                                     917
  ddi_xbuf_get                                                      917
  sd_buf_iodone                                                     917
  sd_destroypkt_for_buf                                             917
  sd_mapblockaddr_iodone                                            917
  sd_return_command                                                 917
  sdintr                                                            917
  xbuf_dispatch                                                     917
  sd_start_cmds                                                    1823
```

To get detailed insight into disk driver operation, each of these functions can be traced in more detail using the fbt provider. The fbt provider can examine the function entry arguments, returning value and time to complete the function. As with stack traces, it is difficult to make much sense of these functions without access to the source code.

Scripts

Table 4-5 summarizes the scripts that follow in this chapter and the providers they use.

Table 4-5 Script Summary

Script	Target	Description	Provider
iolatency.d	I/O	Systemwide I/O latency as a distribution plot	io
disklatency.d	I/O	Measures I/O latency and shows as a distribution plot by device	io
iotypes.d	I/O	Measures I/O latency by type of I/O	io
rwtime.d	I/O	Shows read and write I/O times	io
bitesize.d	I/O	Shows disk I/O sizes as a distribution plot	io
seeksize.d	I/O	Shows disk I/O seek distances as a distribution plot	io
iosnoop	I/O	Traces disk I/O live with various details	io
iotop	I/O	Summarizes disk I/O and refresh screen	io
iopattern	I/O	Shows disk I/O statistics including %random	io
geomiosnoop.d	I/O	Traces GEOM I/O requests (FreeBSD)	fbt
sdqueue.d	SCSI	Shows I/O wait queue times as a distribution plot by device	fbt, sdt
sdretry.d	SCSI	A status tool for SCSI retries	fbt
scsicmds.d	SCSI	Frequency count SCSI commands, with descriptions	fbt
scsilatency.d	SCSI	Summarizes SCSI command latency by type and result	fbt
scsirw.d	SCSI	Shows various SCSI read/write/sync statistics, including bytes	fbt
scsireasons.d	SCSI	Shows SCSI I/O completion reasons and device names	fbt
scsi.d	SCSI	Traces SCSI I/O live with various details or generates reports	fbt
satacmds.d	SATA	Frequency count SATA commands, with descriptions	fbt
satarw.d	SATA	Shows various SATA read/write/sync statistics, including bytes	fbt

Table 4-5 Script Summary (*Continued*)

Script	Target	Description	Provider
`satareasons.d`	SATA	Shows SATA I/O completion reasons and device names	fbt
`satalatency.d`	SATA	Summarizes SATA command latency by type and result	fbt
`idelatency.d`	IDE	Summarizes IDE command latency by type and result	fbt
`iderw.d`	IDE	Shows IDE read/write/sync statistics, including bytes	fbt
`ideerr.d`	IDE	Shows IDE command completion reasons with errors	fbt
`mptsasscsi.d`	SAS	Shows SAS SCSI commands with SCSI and mpt details	fbt
`mptevents.d`	SAS	Traces special mpt SAS events with details	sdt, fbt
`mptlatency.d`	SAS	Shows mpt SCSI command times as a distribution plot	sdt

The fbt and sdt providers are considered "unstable" interfaces, because they instrument a specific operating system or application version. For this reason, scripts that use these providers may require changes to match the version of the software you are using. These scripts have been included here as examples of D programming and of the kind of data that DTrace can provide for each of these topics. See Chapter 12 for more discussion about using the fbt provider.

io Provider Scripts

This is a stable interface for tracing I/O, and it is usually the first provider you should try using when analyzing disk I/O. Functionally, it can be represented as in Figure 4-3.

This high-level diagram can be referenced in the "Capabilities" section.

The I/O provider scripts may also trace NFS client I/O, if your io provider version supports it (currently only on Solaris).

iolatency.d

This shows disk I/O latency as a distribution plot. Since this uses the io provider, it includes NFS client back-end I/O on Solaris.

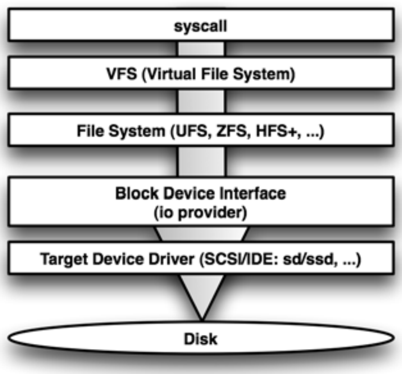


Figure 4-3 Basic I/O stack

Script

I/O latency is calculated as the time between `io:::start` and `io:::done`. Since these events will occur in different threads (`io:::done` fires as part of the I/O completion interrupt), they cannot be associated using thread-local variables, because you would usually perform for latency calculations in DTrace. Instead, an associative array is used, keyed on a unique ID for the I/O.

About arg0. The unique ID chosen to associate I/O events is the `buf_t` pointer before it is translated into the io provider's `bufinfo_t args[0]` (see `/usr/lib/dtrace/io.d`). This is available as `arg0`, a `uint64_t`. Using `arg0` as a unique ID in the io provider is not described in the DTrace Guide, but it is the easiest unique ID available, because the `buf_t` pointer doesn't change between `io:::start` and `io:::done`. At least this is true on current versions of Solaris and Mac OS X. If it changes in the future, the use of `arg0` in this script (and others in this chapter) will need to change to pick a different unique identifier such as what `iosnoop` uses: `args[0]->b_edev` and `args[0]->b_blkno`—both of which are stable members.

```
#!/usr/sbin/dtrace -s

io:::start
{
     start[arg0] = timestamp;
}

io:::done
/start[arg0]/
{
    @time["disk I/O latency (ns)"] = quantize(timestamp - start[arg0]);
```

```
        start[arg0] = 0;
}
```

Script iolatency.d

Example

This was executed on a Mac OS X laptop while a disk read workload was executing. The distribution plot shows bimodal behavior: The most frequent group of I/O completed between 131 us and 524 us, and another group completed between 1 ms and 4 ms. While tracing, 15 I/Os reached the 67 ms to 134 ms range, which is slow even for a laptop disk and may be evidence of queueing (the use of more DTrace can determine this).

```
# iolatency.d
dtrace: script 'iolatency.d' matched 2 probes
^C

  disk I/O latency (ns)
           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |@                                        26
          131072 |@@@@@@@@@@@                              236
          262144 |@@@@@@@@                                 171
          524288 |@@@                                      65
         1048576 |@@                                       52
         2097152 |@@@@                                     88
         4194304 |@@@                                      68
         8388608 |@@                                       52
        16777216 |@@                                       40
        33554432 |@                                        28
        67108864 |@                                        15
       134217728 |                                         0
```

disklatency.d

The `disklatency.d` script measures the time for I/O to complete by device and shows the times as a distribution plot. This is particularly useful for finding disk devices that have occasional slow I/O, which may not be easy to identify via other metrics such as average service time.

Script

This is an enhanced version of `iolatency.d`:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
```

continues

```
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

io:::start
{
        start_time[arg0] = timestamp;
}

io:::done
/this->start = start_time[arg0]/
{
        this->delta = (timestamp - this->start) / 1000;
        @[args[1]->dev_statname, args[1]->dev_major,
                     args[1]->dev_minor] =
            quantize(this->delta);
        start_time[arg0] = 0;
}

dtrace:::END
{
        printa("   %s (%d,%d), us:\n%@d\n", @);
}
```

Script disklatency.d

The predicate for `io:::done` sets the clause-local variable `this->start` and checks that it has a nonzero value. This wasn't strictly necessary; the script could have tested `start_time[arg0]` directly in the predicate as `iolatency.d` did; this just demonstrates a different coding style.

Examples

The examples that follow demonstrate the use of the `disklatency.d` script on a server system running Solaris and a Mac OS X desktop.

Disk I/O on a Solaris Server. The title before each distribution plot includes the device `statname` (if available), device major and minor numbers, and time units (in microseconds, or *us*). Comparing the two disks shows that sd112 is a little faster than sd118, which has more I/O returning in the 32-ms to 65-ms range. The difference here is small and might well be because of the applied workload rather than properties of the disk. Some `nfs` I/O was also caught, mostly with an I/O time between 8 ms and 16 ms.

```
solaris# disklatency.d
Tracing... Hit Ctrl-C to end.
^C

   sd112 (227,7168), us:

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |                                         3
```

```
           2048 |@                                        16
           4096 |@@@@                                     41
           8192 |@@@@@@                                   66
          16384 |@@@@@@@@@@@@@@@@@@@@@@@                  259
          32768 |@@@@@                                    51
          65536 |@                                        14
         131072 |                                         0

  sd118 (227,7552), us:

          value  ------------- Distribution ------------- count
            512 |                                         0
           1024 |@                                        6
           2048 |@@@                                      29
           4096 |@@@@@                                    47
           8192 |@@@@@@@@                                 80
          16384 |@@@@@@@@@@@@                             120
          32768 |@@@@@@@@@@                               101
          65536 |@                                        13
         131072 |                                         0

  nfs1 (309,1), us:

          value  ------------- Distribution ------------- count
           1024 |                                         0
           2048 |@@                                       2
           4096 |@@                                       2
           8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     48
          16384 |                                         0
          32768 |                                         0
          65536 |@                                        1
         131072 |                                         0
[...]
```

Disk I/O on Mac OS X. Here, disk I/O is returning very quickly, often less than 1 ms (which may be because of hits on an on-disk cache). The device name was not available and is printed as `??`.

```
macosx# disklatency.d
Tracing... Hit Ctrl-C to end.
^C
  ?? (14,2), us:

          value  ------------- Distribution ------------- count
             16 |                                         0
             32 |                                         1
             64 |@@@@@@@                                  924
            128 |@@@@@@@@@@@@@                            1744
            256 |@                                        146
            512 |@@@@@@@@                                 1146
           1024 |@@                                       210
           2048 |@@@                                      379
           4096 |@@@@                                     493
           8192 |@@                                       291
          16384 |@                                        87
          32768 |                                         1
          65536 |                                         2
         131072 |                                         2
         262144 |                                         0
```

iotypes.d

The previous script printed I/O latency by disk device; the `iotypes.d` script prints I/O latency by type of I/O, from the `b_flags` member of `args[0]`, which is a pointer to a `buf_t` structure.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           printf("Tracing... Hit Ctrl-C to end.\n");
8   }
9
10  io:::start
11  {
12          start_time[arg0] = timestamp;
13  }
14
15  io:::done
16  /this->start = start_time[arg0]/
17  {
18          this->delta = (timestamp - this->start) / 1000;
19          this->type = args[0]->b_flags & B_READ ? "read" : "write";
20          this->type = args[0]->b_flags & B_PHYS ?
21              strjoin("phys-", this->type) : this->type;
22          this->type = args[0]->b_flags & B_ASYNC ?
23              strjoin("async-", this->type) : this->type;
24          this->pageio = args[0]->b_flags & B_PAGEIO ? "yes" : "no";
25          this->error = args[0]->b_error != 0 ?
26              strjoin("Error:", lltostr(args[0]->b_error)) : "Success";
27
28          @num[this->type, this->pageio, this->error] = count();
29          @average[this->type, this->pageio, this->error] = avg(this->delta);
30          @total[this->type, this->pageio, this->error] = sum(this->delta);
31
32          start_time[arg0] = 0;
33  }
34
35  dtrace:::END
36  {
37          normalize(@total, 1000);
38          printf("\n  %-18s %6s %10s %11s %11s %12s\n", "TYPE", "PAGEIO",
39              "RESULT", "COUNT", "AVG(us)", "TOTAL(ms)");
40          printa("  %-18s %6s %10s %@11d %@11d %@12d\n", @num, @average, @total);
41  }
```

Script iotypes.d

The type description is constructed between lines 19 and 23 by testing each flag using bitwise-AND (`&`) in ternary operators (`a ? b : c`). If the flag is present, a string description is included in the `this->type` variable by use of the built-in `strjoin()` function. The end result is a single string that describes the flags, which is printed in the `TYPE` column on line 40.

Example

A storage server was performing a read-intensive workload when one of its disks was removed. The `iotypes.d` script identified the error, along with the latency caused:

```
solaris# iotypes.d
Tracing... Hit Ctrl-C to end.
^C

  TYPE                 PAGEIO     RESULT        COUNT      AVG(us)     TOTAL(ms)
  read                     no   Error:14            4     30461117        121844
  phys-write               no    Success            5         8696            43
  phys-read                no    Error:5            6          216             1
  phys-read                no    Success         3479         7163         24923
  write                    no    Success         4409        16568         73048
  read                     no    Success       395020        23209       9168382
```

There were four reads that returned error 14 ("Bad address," from `/usr/include/sys/errno.h`) and six physical reads that returned error 5 ("I/O error"). The bad address errors had an average latency of 30 seconds, which could cause serious performance issues for stalled applications. This time of 30 seconds didn't originate from the disk, which was removed but from a lower-level driver—mostly likely SCSI. The inner working of SCSI can also be examined using DTrace; see the examples later in this chapter.

rwtime.d

A useful metric provided by the `iostat(1M)` disk statistic tool is the I/O service time (`svc_t` on Solaris, `msps` on Mac OS X). It includes both read and write I/O in the same average, which may be undesirable because of the following reasons.

- I/O by default to many file systems (including UFS and ZFS) will cause synchronous read I/O (because the application is waiting) and asynchronous write I/O (writes are buffered and flushed later). Because the application does not wait for the write I/O to complete, whether it completes quickly or slowly does not matter; indeed, the kernel may buffer a large group of write I/O together (in ZFS, this is called a *transaction group*), which it flushes to disk all at once. This can cause some of the write I/O to queue for a long time and therefore have high service times. When `iostat(1M)` prints both read and write service times together, an administrator may notice spikes in service time in `iostat(1M)` corresponding to ZFS transaction group syncs and may believe that this may be causing a problem with application performance.
- Flash memory–based, solid-state disks (SSDs) can have considerably different response times for read vs. write I/O.

The previous script, `iolatency.d`, split read and write I/O times into separate metrics. Here we'll take it further and produce distribution plots to examine the I/O latency in more detail.

Script

The `rwtime.d` script measures both read and write I/O latency separately, printing both distribution plots and averages:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

io:::start
{
        start_time[arg0] = timestamp;
}

io:::done
/(args[0]->b_flags & B_READ) && (this->start = start_time[arg0])/
{
        this->delta = (timestamp - this->start) / 1000;
        @plots["read I/O, us"] = quantize(this->delta);
        @avgs["average read I/O, us"] = avg(this->delta);
        start_time[arg0] = 0;
}

io:::done
/!(args[0]->b_flags & B_READ) && (this->start = start_time[arg0])/
{
        this->delta = (timestamp - this->start) / 1000;
        @plots["write I/O, us"] = quantize(this->delta);
        @avgs["average write I/O, us"] = avg(this->delta);
        start_time[arg0] = 0;
}

dtrace:::END
{
        printa("   %s\n%@d\n", @plots);
        printa(@avgs);
}
```

Script rwtime.d

The keys for the `@plots` and `@avgs` aggregations are descriptive strings that include the units, "us" (microseconds), with all output generated. This helps readability and makes the output self-descriptive.

Example

This was executed on Solaris with ZFS performing a disk scrub:

```
solaris# rwtime.d
Tracing... Hit Ctrl-C to end.
^C
   write I/O, us

           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |                                         2
              32 |                                         0
              64 |                                         2
             128 |                                         12
             256 |@                                        22
             512 |@@                                       43
            1024 |@@@@                                     99
            2048 |@                                        29
            4096 |@@@@@                                    126
            8192 |@@@@@                                    133
           16384 |@@@@@                                    132
           32768 |@@@@@@@                                  173
           65536 |@@@@@@@                                  189
          131072 |@@                                       63
          262144 |                                         0
          524288 |                                         3
         1048576 |                                         0

   read I/O, us

           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |                                         8
             256 |                                         98
             512 |@                                        429
            1024 |@@@                                      1197
            2048 |@@@@@@                                   2578
            4096 |@@@@@@@@@@@@@@@                          6216
            8192 |@@@@@@@@@@                               4181
           16384 |@@@@                                     1586
           32768 |@                                        333
           65536 |                                         27
          131072 |                                         0


  average read I/O, us                                          8675
  average write I/O, us                                        42564
```

This output shows a considerable difference between the I/O time for reads and writes. Reads are completing in 8.7 ms on average, while writes are averaging 42.6 ms. The distribution plots allow outliers to be easily identified, which, because of some disk pathologies (errors, retries), can cause I/O to take longer than one second. In the write plot, we see three writes that took between 524 milliseconds and 1 second; 63 of the writes took between 131 and 262 milliseconds. Such outliers can cause serious I/O issues but are difficult to identify from averages alone.

bitesize.d

`bitesize.d` is a script to trace disk I/O events by process name and I/O size. It is from the DTraceToolkit and is available on OpenSolaris in `/opt/DTT` and Mac OS X in /usr/bin. Checking disk I/O size can be useful to ensure that the physical disks are using an optimal size for the workload applied: large I/O for streaming workloads and sizes to match the application's record size for random workloads.

Script

bitesize.d is a simple script; without comments, it's only 20 lines:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       printf("Tracing... Hit Ctrl-C to end.\n");
}

io:::start
{
        this->size = args[0]->b_bcount;
        @Size[pid, curpsinfo->pr_psargs] = quantize(this->size);
}

dtrace:::END
{
        printf("\n%8s  %s\n", "PID", "CMD");
        printa("%8d  %S\n%@d\n", @Size);
}
```

Script bitesize.d

The io:::start probe traces physical disk I/O requests and records the size in bytes in the quantize aggregation @Size, which will be printed as a distribution plot. The aggregation is keyed on the process PID and process name (derived from curpsinfo->pr_psargs). This script uses a clause-local variable and an aggregation, both called size. To differentiate between them, the aggregation was given a capital letter (@Size). This is a matter of taste for the programmer: It is not necessary in D programs, since DTrace will treat this->size and @size as different variables (which they are).

Examples

The following examples demonstrate using bitesize.d under known workloads to determine disk I/O sizes.

Launching Mozilla Firefox. On Mac OS X, bitesize.d was running while the Firefox 3 Web browser was launched. The size of the disk I/O that Firefox caused can be understood in the distribution plot:

```
macosx# bitesize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
    1447  firefox-bin\0

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |                                         8
            1024 |                                         0
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@                 532
            8192 |@@@@@                                    108
           16384 |@@                                       39
           32768 |@@                                       42
           65536 |@@@@@                                    108
          131072 |@                                        30
          262144 |                                         3
          524288 |                                         0
```

Firefox was mostly causing 4KB disk I/O, with only some I/O reaching the 128KB and 256KB ranges. Further investigation with DTrace can identify which I/O (the target file) was 4KB and which I/O was larger. The `iosnoop` tool from the DTraceToolkit, shown later, will provide this information.

Known Test. On Solaris, the `dd(1M)` command was used with a raw disk devices path (`/dev/rdsk`) to cause a known disk I/O workload of 32KB I/Os. This type of simple experiment is recommended whenever possible to double-check script output.

```
solaris# bitesize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
   29245  dd if=/dev/rdsk/c1d0s0 of=/dev/null bs=32k

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |                                         1
            2048 |                                         1
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 7784
           65536 |                                         0
```

Although most of the I/O was in the 32KB to 63KB byte bucket of the distribution plot, as expected, there were also a couple of I/Os between 1KB and 4KB. This is likely to be the `dd(1)` command paging in its own binary from `/usr/bin`, before executing and applying the known workload. Again, further DTrace can confirm.

seeksize.d

The `seeksize.d` script traces disk I/O events by process name and the requested I/O seek distance by application workloads. It is from the DTraceToolkit and is available on OpenSolaris in `/opt/DTT` and Mac OS X in `/usr/bin`. Workloads that cause large disk seeks can incur high I/O latency.

Script

The following is `seeksize.d` without the inline comments:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       printf("Tracing... Hit Ctrl-C to end.\n");
}

self int last[dev_t];

io:::start
/self->last[args[0]->b_edev] != 0/
{
        this->last = self->last[args[0]->b_edev];
        this->dist = (int)(args[0]->b_blkno - this->last) > 0 ?
            args[0]->b_blkno - this->last : this->last - args[0]->b_blkno;
        @Size[pid, curpsinfo->pr_psargs] = quantize(this->dist);
}

io:::start
{
        self->last[args[0]->b_edev] = args[0]->b_blkno +
            args[0]->b_bcount / 512;
}

dtrace:::END
{
        printf("\n%8s  %s\n", "PID", "CMD");
        printa("%8d  %S\n%@d\n", @Size);
}
```

Script seeksize.d

The seek size is calculated as the difference between the disk block addresses of subsequent I/O on the same disk, for the same thread. To do this, the address of each I/O is saved in an associative array keyed on disk so that it can be retrieved for next I/O and the calculation performed. To track the pattern of I/O requests from a single application, the I/O must be from the same thread, which was achieved by making the associative array a thread-local variable: `self->last[]`.

To understand why the thread-local context was necessary, imagine that the associative array was not thread-local (that is, just `last[]`). This would measure the disk seek pattern, regardless of the running application. If two different applications were performing sequential I/O to different areas of the disk, the script would then identify both applications as having performed random disk I/O, which is a consequence of them running at the same time when in fact they are requesting sequential I/O. By making it a thread-local variable, we are able to show what disk seeks occurred because of application *requests*, not as a consequence of other applications running on the system. (If the later is interesting as well, modify the script to measure it as described.)

Examples

Disk seek activity will vary significantly based on the disk I/O load. Sequential disk I/Os will not result in a large number of disk seeks, whereas random disk I/O tends to be dominated by seek activity. The following examples demonstrate this.

Sequential Workload. Here a large file is copied using the `scp(1)` command to a remote host, on Solaris with UFS. The pattern of reading the file from disk should be mostly sequential, if the file system has placed it that way:

```
# seeksize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
   22349  scp /dl/sol-10-b63-x86-v1.iso mars:\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   726
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |@                                        13
              16 |                                         4
              32 |                                         0
              64 |                                         0
             128 |                                         2
             256 |                                         3
             512 |                                         4
            1024 |                                         4
            2048 |                                         3
            4096 |                                         0
```

The large count for zero shows that many of the I/Os did not seek from the previous location of the disk, confirming that this workload is sequential.

Random Workload. Here the `find(1)` command is executed on Solaris/UFS, which will walk directories:

```
solaris# seeksize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
   22399  find /var/sadm/pkg/\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@                            1475
               1 |                                         0
               2 |                                         44
               4 |@                                        77
               8 |@@@                                      286
              16 |@@                                       191
              32 |@                                        154
              64 |@@                                       173
             128 |@@                                       179
             256 |@@                                       201
             512 |@@                                       186
            1024 |@@                                       236
            2048 |@@                                       201
            4096 |@@                                       274
            8192 |@@                                       243
           16384 |@                                        154
           32768 |@                                        113
           65536 |@@                                       182
          131072 |@                                        81
          262144 |                                         0
```

The output shows both a sequential component (1,475 I/Os with a seek value of zero) and a substantial random component (thousands of I/Os).

Running gunzip. A large gzip'd file on Mac OS X in the HFS file system was uncompressed using `gunzip(1)`:

```
macosx# seeksize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
    1651  gunzip\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         292
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |                                         0
            2048 |                                         0
```

```
           4096 |                                         0
           8192 |                                         0
          16384 |                                         0
          32768 |                                         0
          65536 |                                         0
         131072 |                                         0
         262144 |                                         0
         524288 |                                         0
        1048576 |@@@@@@@@                                 73
        2097152 |                                         0
```

Many I/Os are sequential (seek of 0), which can be both the reading of the archive file and the writing of the expanded file. Seventy-three of the I/O events caused an equal-sized seek of more than 1 million disk blocks; it is likely that this occurs whenever `gunzip(1)` switches from reading the source to writing the destination.

iosnoop

> "sudo iosnoop" for slowdowns—the most useful command on OS X?
>
> —Alec Muffet, network security specialist, dropsafe blog

You can use `iosnoop` to trace physical disk I/O events, which are traced using the io provider. Probably the most popular DTrace script written to date, it is from the DTraceToolkit and is available on OpenSolaris in `/opt/DTT` and Mac OS X in `/usr/bin`. The most recent version on Solaris now allows it to trace all io provider events, including NFS client I/O.

It traces disk I/O events that cause disks to physically read or write data. For rotating disks, these events include disk head seek and platter rotation time, which can add significant latency to applications, making these events very interesting for performance analysis.

The following example shows `iosnoop` tracing the disk I/O caused by the `gzip(1)` command compressing the file `source.tar`, on Solaris with the UFS file system:

```
# iosnoop
  UID   PID D    BLOCK   SIZE       COMM PATHNAME
    0 28777 R   310160   4096       bash /usr/bin/gzip
    0 28777 R  3438352   8192       gzip /export/home/brendan/source.tar
    0 28777 R  3438368   8192       gzip /export/home/brendan/source.tar
    0 28777 R  3438384  57344       gzip /export/home/brendan/source.tar
    0 28777 R  3438496  24576       gzip /export/home/brendan/source.tar
    0 28777 R 16627552   8192       gzip <none>
    0 28777 R 16627568  57344       gzip /export/home/brendan/source.tar
    0 28777 R 16627680  57344       gzip /export/home/brendan/source.tar
    0 28777 R 16627792  57344       gzip /export/home/brendan/source.tar
    0 28777 R 16627904  24576       gzip /export/home/brendan/source.tar
    0 28777 W  3632400  57344       gzip /export/home/brendan/source.tar.gz
```

continues

```
     0 28777 R 25165872   8192       gzip <none>
     0 28777 W  3632512  40960       gzip /export/home/brendan/source.tar.gz
     0 28777 W 25180896  24576       gzip /export/home/brendan/source.tar.gz
     0 28777 W 25180944  32768       gzip /export/home/brendan/source.tar.gz
[...]
```

The `COMM` column shows the process that was on-CPU when the disk I/O was called. The first line shows that the `bash` command read 4KB from the `/usr/bin/gzip` binary, followed by I/O called by the `gzip` program. Note the direction of each I/O in the `D` column; gzip begins to read (`R`) from the `source.tar` file and finishes by writing (`W`) to the `.gz` output file. The two lines that show `<none>` for the path name are likely to be when the file system read metadata from disk containing the file system layout (DTrace can confirm this with specific tracing).

Figure 4-4 may help you visualize the flow of I/O from application to disk that `iosnoop` is tracing (refer to the "Capabilities" section for a more complete diagram).

`iosnoop` perhaps should have been called "disksnoop" to emphasize that it is examining *disk* I/O events. Some users have been confused when examining an application performing known reads and writes and finding that `iosnoop` is not tracing those events. This is because applications usually do not perform disk I/O directly; rather, they perform I/O to a file system (such as ZFS), which in turn performs I/O to disks. File systems may return read I/O from their in-DRAM caches and buffer write I/O for later asynchronous flushing—both of which cause application I/O to *not* trigger an immediate disk I/O, which means the `io:::start` probe will not fire. `io:::start` fires only for physical disk I/Os.

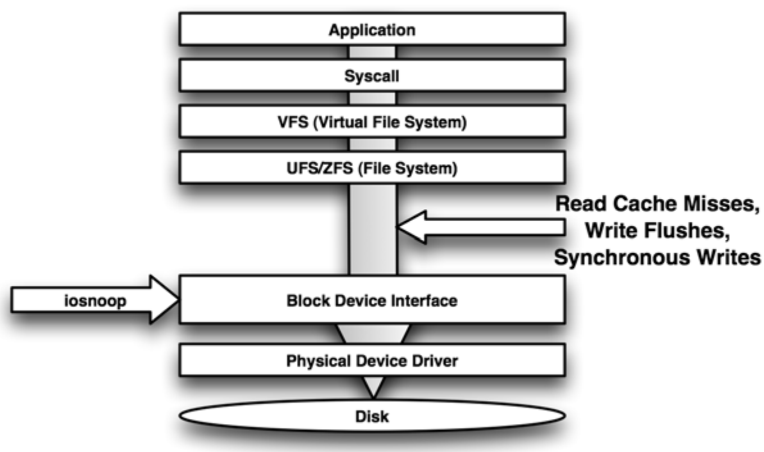


Figure 4-4 Application I/O stack

The iosnoop script allows a variety of options to customize the output:

```
# iosnoop -h
USAGE: iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
               [-m mount_point] [-n name] [-p PID]
       iosnoop          # default output
                -a      # print all data (mostly)
                -A      # dump all data, space delimited
                -D      # print time delta, us (elapsed)
                -e      # print device name
                -g      # print command arguments
                -i      # print device instance
                -N      # print major and minor numbers
                -o      # print disk delta time, us
                -s      # print start time, us
                -t      # print completion time, us
                -v      # print completion time, string
                -d device       # instance name to snoop
                -f filename     # snoop this file only
                -m mount_point  # this FS only
                -n name         # this process name only
                -p PID          # this PID only
   eg,
        iosnoop -v      # human readable timestamps
        iosnoop -N      # print major and minor numbers
        iosnoop -m /    # snoop events on file system / only
```

Because of this complexity, the script itself is a DTrace script wrapped in a shell script to process these options. As an example of option usage, the following shows tracing the SSH daemon sshd on Solaris, with disk I/O times:

```
# iosnoop -o -n sshd
DTIME(us)     UID   PID D    BLOCK   SIZE       COMM PATHNAME
5579            0 28870 R  7963696   4096       sshd /usr/lib/gss/mech_krb5.so.1
9564            0 28870 R  3967344   4096       sshd /usr/lib/gss/mech_spnego.so.1
221             0 28870 W    48304   3072       sshd /var/adm/lastlog
479             0 28870 W     2657    512       sshd <none>
17123           0 28870 R   145336   4096       sshd /var/adm/wtmpx
```

The slowest I/O was a read from /var/adm/wtmpx, at 17 milliseconds (17,123 microseconds).

Internals

The bulk of this script is about option processing rather than tracing disk I/O. As an example of how shell scripting can add option processing to a DTrace script, we will explain the entire source of iosnoop.

The first line invokes the Bourne shell:

```
1   #!/bin/sh
2   #
3   # iosnoop - A program to print disk I/O events as they happen, with useful
4   #           details such as UID, PID, filename (if available), command, etc.
5   #           Written using DTrace (Solaris 10 3/05, MacOS X 10.5).
6   #
7   # This is measuring events that have made it past system caches, such as
8   # disk events for local file systems, and network events for remote
9   # file systems (such as NFS.)
10  #
```

These are shell comment lines (beginning with #), so they are not executed. All DTraceToolkit scripts begin with a similar style of block comment, naming the script and providing a short synopsis of its function.

```
11  # $Id: iosnoop 75 2009-09-15 09:06:31Z brendan $
12  #
```

Line 11 includes an identifier tag from Subversion, the source repository for the DTraceToolkit. This shows when this script was last updated (2009-09-15), which is effectively its version number.

```
13  # USAGE:        iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
14  #                       [-m mount_point] [-n name] [-p PID]
15  #
16  #               iosnoop         # default output
17  #
18  #               -a              # print all data (mostly)
19  #               -A              # dump all data, space delimited
20  #               -D              # print time delta, us (elapsed)
21  #               -e              # print device name
22  #               -g              # print command arguments
23  #               -i              # print device instance
24  #               -N              # print major and minor numbers
25  #               -o              # print disk delta time, us
26  #               -s              # print start time, us
27  #               -t              # print completion time, us
28  #               -v              # print completion time, string
29  #               -d device       # instance name to snoop (eg, dad0)
30  #               -f filename     # full pathname of file to snoop
31  #               -m mount_point  # this FS only (will skip raw events)
32  #               -n name         # this process name only
33  #               -p PID          # this PID only
34  #  eg,
35  #               iosnoop -v      # human readable timestamps
36  #               iosnoop -N      # print major and minor numbers
37  #               iosnoop -m /    # snoop events on the root file system only
38  #
39  # FIELDS:
40  #               UID             user ID
41  #               PID             process ID
42  #               PPID            parennt process ID
43  #               COMM            command name for the process
```

```
#                ARGS            argument listing for the process
#                SIZE            size of operation, bytes
#                BLOCK           disk block for the operation (location)
#                STIME           timestamp for the disk request, us
#                TIME            timestamp for the disk completion, us
#                DELTA           elapsed time from request to completion, us
#                DTIME           time for disk to complete request, us
#                STRTIME         timestamp for the disk completion, string
#                DEVICE          device name
#                INS             device instance number
#                D               direction, Read or Write
#                MOUNT           mount point
#                FILE            filename (basename) for io operation
#
```

The previous usage and fields descriptions are also in the man page for `iosnoop`.

```
# NOTE:
# - There are two different delta times reported. -D prints the
#   elapsed time from the disk request (strategy) to the disk completion
#   (iodone); -o prints the time for the disk to complete that event
#   since it's last event (time between iodones), or, the time to the
#   strategy if the disk had been idle.
# - When filtering on PID or process name, be aware that poor disk event
#   times may be due to events that have been filtered away, for example
#   another process that may be seeking the disk heads elsewhere.
#
```

The previous are some important notes for understanding disk event times.

```
# SEE ALSO: BigAdmin: DTrace, http://www.sun.com/bigadmin/content/dtrace
#           Solaris Dynamic Tracing Guide, http://docs.sun.com
#           DTrace Tools, http://www.brendangregg.com/dtrace.html
#
# COPYRIGHT: Copyright (c) 2009 Brendan Gregg.
#
# CDDL HEADER START
#
#  The contents of this file are subject to the terms of the
#  Common Development and Distribution License, Version 1.0 only
#  (the "License").  You may not use this file except in compliance
#  with the License.
#
#  You can obtain a copy of the license at Docs/cddl1.txt
#  or http://www.opensolaris.org/os/licensing.
#  See the License for the specific language governing permissions
#  and limitations under the License.
#
# CDDL HEADER END
#
```

The intent to share this script openly is made clear by the use of a standard open source license. This has encouraged the inclusion of this and other scripts in operating systems such as OpenSolaris and Mac OS X.

```
# 12-Mar-2004   Brendan Gregg   Created this, build 51.
# 23-May-2004      "      "     Fixed mntpt bug.
# 10-Oct-2004      "      "     Rewritten to use the io provider, build 63.
# 04-Jan-2005      "      "     Wrapped in sh to provide options.
# 08-May-2005      "      "     Rewritten for perfromance.
# 15-Jul-2005      "      "     Improved DTIME calculation.
# 25-Jul-2005      "      "     Added -p, -n. Improved code.
# 17-Sep-2005      "      "     Increased switchrate.
# 15-Sep-2009      "      "     Removed genunix for both MacOS X and NFS.
#
```

That was some script history.

```


##############################
# --- Process Arguments ---
#

### default variables
opt_dump=0; opt_device=0; opt_delta=0; opt_devname=0; opt_file=0; opt_args=0;
opt_mount=0; opt_start=0 opt_end=0; opt_endstr=0; opt_ins=0; opt_nums=0
opt_dtime=0; filter=0; device=.; filename=.; mount=.; pname=.; pid=0
opt_name=0; opt_pid=0

```

Now shell scripting begins with initializing the variables that will be used to process the command-line options.

```
### process options
while getopts aAd:Def:ghim:Nn:op:stv name
do
        case $name in
        a)      opt_devname=1; opt_args=1; opt_endstr=1; opt_nums=1 ;;
        A)      opt_dump=1 ;;
        d)      opt_device=1; device=$OPTARG ;;
        D)      opt_delta=1 ;;
        e)      opt_devname=1 ;;
        f)      opt_file=1; filename=$OPTARG ;;
        g)      opt_args=1 ;;
        i)      opt_ins=1 ;;
        N)      opt_nums=1 ;;
        n)      opt_name=1; pname=$OPTARG ;;
        o)      opt_dtime=1 ;;
        p)      opt_pid=1; pid=$OPTARG ;;
        m)      opt_mount=1; mount=$OPTARG ;;
        s)      opt_start=1 ;;
        t)      opt_end=1 ;;
        v)      opt_endstr=1 ;;
        h|?)    cat <<-END >&2
                USAGE: iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
                               [-m mount_point] [-n name] [-p PID]
                       iosnoop          # default output
                                -a      # print all data (mostly)
                                -A      # dump all data, space delimited
                                -D      # print time delta, us (elapsed)
                                -e      # print device name
```

```
                                -g      # print command arguments
                                -i      # print device instance
                                -N      # print major and minor numbers
                                -o      # print disk delta time, us
                                -s      # print start time, us
                                -t      # print completion time, us
                                -v      # print completion time, string
                                -d device       # instance name to snoop
                                -f filename     # snoop this file only
                                -m mount_point  # this FS only
                                -n name         # this process name only
                                -p PID          # this PID only
                   eg,
                        iosnoop -v      # human readable timestamps
                        iosnoop -N      # print major and minor numbers
                        iosnoop -m /    # snoop events on file system / only
                END
                exit 1
        esac
done

```

The `while getopts` loop processes the command-line options. By the time this loop finishes, variables beginning with `opt_` have been set to record which command-line options were used. Any arguments to options are stored in their own variable names (device, filename, and so on).

```
### option logic
if [ $opt_dump -eq 1 ]; then
        opt_delta=0; opt_devname=0; opt_args=2; opt_start=0;
        opt_end=0; opt_endstr=0; opt_nums=0; opt_ins=0; opt_dtime=0
fi
```

If the dump option was used with other options that print the same fields, this ignores the other redundant options by setting their variables to 0.

```
if [ $opt_device -eq 1 -o $opt_file -eq 1 -o $opt_mount -eq 1 -o \
    $opt_name -eq 1 -o $opt_pid -eq 1 ]; then
        filter=1
fi

```

A single variable `filter` is used to track whether the output is filtered by any other option.

```
#################################
# --- Main Program, DTrace ---
#
/usr/sbin/dtrace -n '
```

To keep the script simple to follow, the first half is shell scripting, and the second half is an inline DTrace script. Line 173 is the boundary between the two. It executes `/usr/sbin/dtrace` and finishes with a single quote character (`'`), which feeds the lines that follow to `dtrace(1M)` instead of processing them in the shell.

```
174   /*
175    * Command line arguments
176    */
177   inline int OPT_dump    = '$opt_dump';
178   inline int OPT_device  = '$opt_device';
179   inline int OPT_delta   = '$opt_delta';
180   inline int OPT_devname = '$opt_devname';
181   inline int OPT_file    = '$opt_file';
182   inline int OPT_args    = '$opt_args';
183   inline int OPT_ins     = '$opt_ins';
184   inline int OPT_nums    = '$opt_nums';
185   inline int OPT_dtime   = '$opt_dtime';
186   inline int OPT_mount   = '$opt_mount';
187   inline int OPT_start   = '$opt_start';
188   inline int OPT_pid     = '$opt_pid';
189   inline int OPT_name    = '$opt_name';
190   inline int OPT_end     = '$opt_end';
191   inline int OPT_endstr  = '$opt_endstr';
192   inline int FILTER      = '$filter';
193   inline int PID         = '$pid';
194   inline string DEVICE   = "'$device'";
195   inline string FILENAME = "'$filename'";
196   inline string MOUNT    = "'$mount'";
197   inline string NAME     = "'$pname'";
198
```

Everything between the single quote characters in the previous code is shell context; everything else is DTrace script context. This allows the shell option variables to be passed to the DTrace script. It's made possible by staggering the pairs of single forward quote characters; for example, one pair begins on line 173 and ends on line 177; the next pair begins on line 177 and ends on line 178. Everything between the staggered pairs is not processed by the shell and is passed directly to `/usr/sbin/dtrace`. Shell option variables that start with `$opt_` become DTrace variables that start with `OPT_`.

```
199   #pragma D option quiet
200   #pragma D option switchrate=10hz
201
```

The quiet pragma stops DTrace from printing its default output, because `iosnoop` will print customized output. Setting `switchrate` to 10 Hertz makes the output of `iosnoop` appear more rapidly, rather than printing at the default rate of 1 Hertz.

```
/*
 * Print header
 */
dtrace:::BEGIN
{
        last_event[""] = 0;

        /* print optional headers */
        OPT_start   ? printf("%-14s ","STIME(us)")   : 1;
        OPT_end     ? printf("%-14s ","TIME(us)")    : 1;
        OPT_endstr  ? printf("%-20s ","STRTIME") : 1;
        OPT_devname ? printf("%-7s ","DEVICE")   : 1;
        OPT_ins     ? printf("%-3s ","INS")      : 1;
        OPT_nums    ? printf("%-3s %-3s ","MAJ","MIN") : 1;
        OPT_delta   ? printf("%-10s ","DELTA(us)")   : 1;
        OPT_dtime   ? printf("%-10s ","DTIME(us)")   : 1;

        /* print main headers */
        OPT_dump ?
            printf("%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s\n",
            "TIME", "STIME", "DELTA", "DEVICE", "INS", "MAJ", "MIN", "UID",
            "PID", "PPID", "D", "BLOCK", "SIZE", "MOUNT", "FILE", "PATH",
            "COMM","ARGS") :
            printf("%5s %5s %1s %8s %6s ", "UID", "PID", "D", "BLOCK", "SIZE");
        OPT_args == 0 ? printf("%10s %s\n", "COMM", "PATHNAME") : 1;
        OPT_args == 1 ? printf("%28s %s\n", "PATHNAME", "ARGS") : 1;
}
```

This prints the output header, naming the columns. Depending on which option was used, different column headers will be printed. Since DTrace doesn't currently have `if` statements, ternary operators (`a ? b : c`) are used to either print a column header using `printf()` or to do nothing by including a 1 in the script, which is ignored.

```
/*
 * Check event is being traced
 */
io:::start
{
        /* default is to trace unless filtering, */
        self->ok = FILTER ? 0 : 1;

        /* check each filter, */
        (OPT_device == 1 && DEVICE == args[1]->dev_statname)? self->ok = 1 : 1;
        (OPT_file == 1 && FILENAME == args[2]->fi_pathname) ? self->ok = 1 : 1;
        (OPT_mount == 1 && MOUNT == args[2]->fi_mount) ? self->ok = 1 : 1;
        (OPT_name == 1 && NAME == execname) ? self->ok = 1 : 1;
        (OPT_pid == 1 && PID == pid) ? self->ok = 1 : 1;
}
```

The `self->ok` variable records whether to trace this event. If no filtering is used, it is set to 1 to trace all events. Otherwise, it is set to 1 only if the filtering option is satisfied, by a series of ternary operator tests.

```
/*
 * Reset last_event for disk idle -> start
 * this prevents idle time being counted as disk time.
 */
io:::start
/! pending[args[1]->dev_statname]/
{
        /* save last disk event */
        last_event[args[1]->dev_statname] = timestamp;
}
```

Normally the `last_event` time for a disk is set only when it completes an event, `io:::done`, and is used to calculate the time the disk spent processing a single I/O by taking the delta time between events. However, if the disk is idle and has no pending I/O, then that delta time includes idle time, which is not what we want to measure. To prevent counting idle time, the `last_event` time is reset to the start of any disk I/O if the disk was idle.

```
/*
 * Store entry details
 */
io:::start
/self->ok/
{
        /* these are used as a unique disk event key, */
        this->dev = args[0]->b_edev;
        this->blk = args[0]->b_blkno;

        /* save disk event details, */
        start_uid[this->dev, this->blk] = uid;
        start_pid[this->dev, this->blk] = pid;
        start_ppid[this->dev, this->blk] = ppid;
        start_args[this->dev, this->blk] = (char *)curpsinfo->pr_psargs;
        start_comm[this->dev, this->blk] = execname;
        start_time[this->dev, this->blk] = timestamp;
```

The `io:::start` probe often fires in the same context as the requesting process, so we'd like to save various details about the current process (such as `uid`, `pid`, and `execname`) and refer to them later when needed. We want to be able to print `firefox`, if the requesting application was Firefox, for example.

Important points to note here are the following.

Whether `io:::start` fires in the context of the requesting process depends on the I/O type and any file system involved. Consider the following.

– **reads**: These are likely to fire in the same context of the application, provided that a file system doesn't sleep the application thread before issuing the disk I/O. If file systems begin to do this, `iosnoop` can be modified to trace the `read()` system call and thread context switching so that it can detect whether the application thread leaves the CPU before the disk I/O event and, if so, use process information from the syscall context.

– **prefetch/read-ahead reads**: These occur when a file system detects sequential access and requests data ahead of where the application is currently reading, to improve performance by caching data early. Whether the application is still on-CPU during these I/O requests is up to file system implementation: For UFS it is; for ZFS it usually isn't. `iosnoop` on ZFS may identify the requesting application as `sched` (kernel), instead of the process (which you could argue is correct, because it is the file system/kernel requesting that I/O, not the application—yet). It is always possible—though sometimes difficult—to identify the process using DTrace, because it would require special casing for different file system types.

– **writes**: Depending on the file system, these are often buffered and flushed to disk sometime later, long after the application thread has left CPU. `iosnoop` may identify the process as `sched` (kernel), because it is a kernel thread collecting and flushing these modified file system pages to disk.

– **synchronous writes**: These are likely to fire in the context of the application.

`iosnoop` does what it can using the stable io provider. If identifying the application (and not the kernel) for all types of I/O is important, custom DTrace scripts can be written for the file system type used. The syscall provider is useful for associating disk I/O to applications, since all application-driven disk I/Os originate as system calls.[7] This approach may require two steps. First, determine which system calls are being used (`read(2)`, `pread(2)`, `readv(2)`, and so on). Second, create a script that enables probes at those system calls. If the fbt provider or sdt probes are used, the scripts will not be stable and may require maintenance whenever the file system driver is upgraded.

The process details are saved in associative arrays, `start_uid[]`, and so on. We'd like to save various details on the `io:::start` probe and refer to them in the `io:::done` probe. This is often achieved in DTrace using thread-local variables (`self->`). However, this will not work here because the `io:::done` probe will fire when the disk sends an interrupt, at which point whatever thread was on-CPU during `io:::start` will have long since left, and the thread context will have changed. So, instead of using the thread to associate

7. The exception is the use of `mmap(2)`, which will generate disk reads and writes as application code reads and will modify the memory segment allocated for the mmap'd file.

the start event with the done event via a thread-local variable, we use an associative array instead, keyed on something that will be the same for both `io:::start` and `io:::done`. This is a two-key pair consisting of the disk device ID and the block ID, which identifies the location on the disk this I/O is for and is the same for both `io:::start` and `io:::done`. This key could even be simplified to just `arg0`, as explained in `iolatency.d`.

```
275         /* increase disk event pending count */
276         pending[args[1]->dev_statname]++;
277
```

The `pending[]` associative array tracks how many outstanding disk I/O events there are for each disk. The disks are uniquely identified by their `dev_statname`. For `io:::start`, pending is incremented; for `io:::done`, pending is decremented. By tracking the number of outstanding I/Os, the idle state can be identified on line 251.

```
278         self->ok = 0;
279 }
280
```

The `self->ok` variable is reset for this thread.

```
281 /*
282  * Process and Print completion
283  */
284 io:::done
285 /start_time[args[0]->b_edev, args[0]->b_blkno]/
286 {
```

The predicate here checks that the `start_time` was seen, which will be the case for all I/O except for those in-flight when `iosnoop` was first executed.

```
287         /* decrease disk event pending count */
288         pending[args[1]->dev_statname]--;
289
290         /*
291          * Process details
292          */
293
294         /* fetch entry values */
295         this->dev = args[0]->b_edev;
296         this->blk = args[0]->b_blkno;
297         this->suid = start_uid[this->dev, this->blk];
298         this->spid = start_pid[this->dev, this->blk];
```

```
        this->sppid = start_ppid[this->dev, this->blk];
        self->sargs = (int)start_args[this->dev, this->blk] == 0 ?
            "" : start_args[this->dev, this->blk];
        self->scomm = start_comm[this->dev, this->blk];
        this->stime = start_time[this->dev, this->blk];
        this->etime = timestamp; /* endtime */
```

Various bits of information are fetched and stored in clause-local variables (`this->`), because they are used only in this block of code for `io:::done`. An exception to this is string data types, which are saved as thread-local variables (`self->`), only because earlier versions of DTrace didn't allow strings as clause-local variables.

```
        this->delta = this->etime - this->stime;
        this->dtime = last_event[args[1]->dev_statname] == 0 ? 0 :
            timestamp - last_event[args[1]->dev_statname];

```

Note the different ways I/O latency can be calculated. `this->delta` (printed with `-D`) is the time from `io:::start` to `io:::done`, and it reflects the latency for this I/O to complete. Although this is a simple and useful metric, bear in mind that many I/O devices can service multiple I/Os simultaneously. Disks are often sent multiple I/O requests that they can queue, reorder, and access with one sweep of the heads (sometimes called *elevator seeking*). Since the disk is servicing multiple I/Os simultaneously, it is possible that the reported delta times from a disk during a one-second interval can add up to much more than one second.

`this->dtime` (printed with `-o`) calculates I/O latency as the time from the last disk completion event to the current disk completion event. This is the time it took the disk to seek and service the current I/O, excluding other I/O that was queued. Adding these times during a one-second interval should not sum to more than one second. Put another way, the delta time is the latency suffered by the application waiting on that I/O. The disk time is the latency suffered by the disk to service that I/O.

```
        /* memory cleanup */
        start_uid[this->dev, this->blk]  = 0;
        start_pid[this->dev, this->blk]  = 0;
        start_ppid[this->dev, this->blk] = 0;
        start_args[this->dev, this->blk] = 0;
        start_time[this->dev, this->blk] = 0;
        start_comm[this->dev, this->blk] = 0;
        start_rw[this->dev, this->blk]   = 0;

```

This data is no longer needed and is cleared from the associative arrays. This data was stored in seven associative arrays, and not one associative array with a seven-member struct as the value, because there is currently no way to clear such a seven-member struct.[8]

```
/*
 * Print details
 */

/* print optional fields */
OPT_start   ? printf("%-14d ", this->stime/1000) : 1;
OPT_end     ? printf("%-14d ", this->etime/1000) : 1;
OPT_endstr  ? printf("%-20Y ", walltimestamp) : 1;
OPT_devname ? printf("%-7s ", args[1]->dev_statname) : 1;
OPT_ins     ? printf("%3d ", args[1]->dev_instance) : 1;
OPT_nums    ? printf("%3d %3d ",
    args[1]->dev_major, args[1]->dev_minor) : 1;
OPT_delta   ? printf("%-10d ", this->delta/1000) : 1;
OPT_dtime   ? printf("%-10d ", this->dtime/1000) : 1;

```

The optional fields are printed in ternary statements, in the same order as the column headers were printed.

```
/* print main fields */
OPT_dump ?
    printf("%d %d %d %s %d %d %d %d %d %d %s %d %d %s %s %s %s %S\n",
    this->etime/1000, this->stime/1000, this->delta/1000,
    args[1]->dev_statname, args[1]->dev_instance, args[1]->dev_major,
    args[1]->dev_minor, this->suid, this->spid, this->sppid,
    args[0]->b_flags & B_READ ? "R" : "W",
    args[0]->b_blkno, args[0]->b_bcount, args[2]->fi_mount,
    args[2]->fi_name, args[2]->fi_pathname, self->scomm, self->sargs) :
    printf("%5d %5d %1s %8d %6d ",
    this->suid, this->spid, args[0]->b_flags & B_READ ? "R" : "W",
    args[0]->b_blkno, args[0]->b_bcount);
```

Look for the `:` in the previous ternary operator. If `OPT_dump` is set (`-A`), everything is printed space delimited; otherwise, the default columns are printed.

```
OPT_args == 0 ? printf("%10s %s\n", self->scomm, args[2]->fi_pathname)
    : 1;
OPT_args == 1 ? printf("%28s %S\n",
    args[2]->fi_pathname, self->sargs) : 1;

/* save last disk event */
last_event[args[1]->dev_statname] = timestamp;
```

8. CR 6411981 says "need a way to unallocate struct dynamic variables."

The last disk event time stamp for calculating disk times is stored for this disk.

```

        /* cleanup */
        self->scomm = 0;
        self->sargs = 0;
```

The clause-local variables were cleared automatically; these thread-local variables need to be cleared explicitly.

```
}

/*
 * Prevent pending from underflowing
 * this can happen if this program is started during disk events.
 */
io:::done
/pending[args[1]->dev_statname] < 0/
{
        pending[args[1]->dev_statname] = 0;
}
```

Should `iosnoop` be executed during several I/Os to a disk, the `io:::done` events will be seen, but the `io:::start` events will not, leading to a negative value for pending. If this happens, pending is reset to zero.

```
'
```

The last line has a single forward quote, which finishes quoting the DTrace script fed into the `/usr/sbin/dtrace` command.

Examples

In the examples that follow, the use of the `iosnoop` script is demonstrated showing various options available for observing different dimensions of your disk I/O load.

Disk Queueing. The following shows four different time measurements taken while tracing a `tar` archive command on Solaris, UFS:

```
solaris# iosnoop -Dots
STIME(us)      TIME(us)       DELTA(us)  DTIME(us) UID  PID D    BLOCK   SIZE     COMM PATHNAME
3323037572603  3323037577515  4912       4919        0  29102 R   493680  16384    tar  /root/perl/perl
3323037572686  3323037581099  8413       3584        0  29102 R   496368  16384    tar  /root/perl/perl
3323037572726  3323037581258  8532       158         0  29102 W   166896   8192    tar  /root/perl.tar
3323037572745  3323037581394  8649       135         0  29102 W   167296   8192    tar  /root/perl.tar
3323037572762  3323037581481  8718       87          0  29102 W   167568   8192    tar  /root/perl.tar
3323037572801  3323037581966  9164       484         0  29102 W   167840  40960    tar  /root/perl.tar
3323037572845  3323037582162  9317       195         0  29102 W     2076   8704    tar  <none>
[...]
```

If you look closely at the start times (STIME), you can see that between the first and last lines (a period of 242 us), seven I/Os were requested. These were queued on the disk and processed in turn, the last completing 9559 us after the first was requested (TIME 3323037582162 to STIME 3323037572603). Time for each I/O from start to done as shown by the delta time (DELTA) column is usually more than 8 ms; however, they were all able to complete in 9 ms because the disk processed several at the same time. The disk time (DTIME) column showed that the slowest I/O took the disk 4.9 ms to service, and the writes (see the D column) returned almost immediately, which is evidence of disk write caching.

Random I/O. The following example shows a spike in disk time for a couple of I/Os, on Solaris, UFS. What might have caused it?

```
solaris# iosnoop -oe
DEVICE  DTIME(us)     UID   PID D    BLOCK   SIZE       COMM PATHNAME
cmdk0   195             0 29127 R   163680  20480        tar /root/sh/sh.new
cmdk0   8240            0 29127 R  2666640  32768        tar /root/sh/sh.new
cmdk0   10816           0 29127 R   163872   8192        tar /root/sh/sh
cmdk0   97              0 29127 R   163888   4096        tar /root/sh/sh
cmdk0   181             0 29127 R   163904  16384        tar /root/sh/sh
```

The disk device (DEVICE) column and disk block address (BLOCK) column show that this disk seeked from block address 163680 to 2666640 and then back to 163872; these large seeks are likely to be the reason for the longer disk times. File systems usually place data to avoid large seeks, if possible, to improve performance.

What Is sched and Why <none>? Here iosnoop is executed on a Solaris server with a ZFS file system, as a tar(1) archive command is executed:

```
solaris# iosnoop
  UID   PID D    BLOCK   SIZE       COMM PATHNAME
  100 29166 R 184306688 131072        tar <none>
  100 29166 R 184306944 131072        tar <none>
  100 29166 R 184307200 131072        tar <none>
  100 29166 R 184307456 131072        tar <none>
  100 29166 R 184307712 131072        tar <none>
    0     0 R 184307968 131072      sched <none>
```

```
     0     0 R 184308224 131072      sched <none>
     0     0 R 184308480 131072      sched <none>
     0     0 W 282938086   8704      sched <none>
     0     0 W 137834280   8704      sched <none>
     0     0 W 137834297   3584      sched <none>
     0     0 W 282342144 131072      sched <none>
     0     0 W 282343424 131072      sched <none>
[...]
```

First, `sched` is the scheduler—the kernel. It appears so frequently because ZFS uses asynchronous threads and pipelining and will call prefetch I/O and flush writes (transaction groups) from pools of ZFS threads in the kernel, not in application context. As for the path name, when ZFS performs disk I/O, multiple I/O requests can be aggregated to improve performance. By the time this I/O reaches disk and fires the `io:::start` or `io:::done` probe, there is a mass of data to read or write, but no single path name (actually, `vnode`) is responsible, so the io provider translator (`/usr/lib/dtrace/io.d`) returns `<none>`.[9] This doesn't affect ZFS functioning but does matter when DTrace is trying to debug that function.

What Is ??? The following shows `iosnoop` run on Mac OS X, as StarOffice is launched:

```
macosx# iosnoop
  UID  PID D    BLOCK  SIZE    COMM PATHNAME
  501  988 R 45293712  4096 launchd ??/MacOS/soffice
  501  988 R 44433744  4096 soffice ??/lib/libuno_sal.dylib.3
    0   22 R 34670448 16384     mds ??/20BAE9D6-DC50-4B38-B8CF-A3D97020E320/.store.db
  501  988 R 44452544 73728 soffice ??/lib/libuno_sal.dylib.3
  501  988 R 44452376 81920 soffice ??/lib/libuno_sal.dylib.3
  501  988 R 44452536  4096 soffice ??/lib/libuno_sal.dylib.3
  501  988 R 44757680  4096 soffice ??/program/libsofficeapp.dylib
  501  988 R 44758496 65536 soffice ??/program/libsofficeapp.dylib
  501  988 R 44758392 49152 soffice ??/program/libsofficeapp.dylib
  501  988 R 44758488  4096 soffice ??/program/libsofficeapp.dylib
  501  988 R 44580232  4096 soffice ??/program/libcomphelp4gcc3.dylib
[...]
```

The path name begins with `??` instead of what should be the mount point path. This originates from the io provider translator (`/usr/lib/dtrace/io.d`), where not all fields are currently available in Mac OS X, which returns `??` when unavailable. These may also be fixed in a future release.

9. This is tracked as CR 6266202: "DTrace io provider doesn't work with ZFS." This hasn't been fixed at the time of writing.

iotop

While `iosnoop` traces and prints each I/O event live, another way to present this data is to print summaries every few seconds. The `iotop` tool does this and will also refresh the screen and print the top several events.[10]

Script

The script gathers data similarly to `iosnoop` and also gathers system load averages by reading and processing the `hp_avenrun` kernel variables. Here's an example on Solaris:

```
/*
 * Print Report
 */
profile:::tick-1sec
/secs == 0/
{
     /* fetch 1 min load average */
     this->load1a  = `hp_avenrun[0] / 65536;
     this->load1b  = ((`hp_avenrun[0] % 65536) * 100) / 65536;
```

The full script can be found in the DTraceToolkit. It is also available as `/usr/bin/iotop` on Mac OS X, where the `` `hp_avenrun `` lines were changed to work with the Mac OS X kernel:

```
this->fscale = `averunnable.fscale;
this->load1a  = `averunnable.ldavg[0] / this->fscale;
this->load1b  = ((`averunnable.ldavg[0] % this->fscale) * 100) / this->fscale;
```

Examples

`iotop` offers several options for tracking which workload processes are generating disk I/O. The examples that follow demonstrate these options.

Usage. Running `iotop` with the `-h` option prints its usage:

```
# iotop -h
USAGE: iotop [-C] [-D|-o|-P] [-j|-Z] [-d device] [-f filename]
             [-m mount_point] [-t top] [interval [count]]

                -C      # don't clear the screen
                -D      # print delta times, elapsed, us
                -j      # print project ID
                -o      # print disk delta times, us
```

10. This is similar to the original `top(1)` tool by William LeFebvre.

```
                -P      # print %I/O (disk delta times)
                -Z      # print zone ID
                -d device       # instance name to snoop
                -f filename     # snoop this file only
                -m mount_point  # this FS only
                -t top  # print top number only
   eg,
        iotop         # default output, 5 second samples
        iotop 1       # 1 second samples
        iotop -P      # print %I/O (time based)
        iotop -m /    # snoop events on file system / only
        iotop -t 20   # print top 20 lines only
        iotop -C 5 12 # print 12 x 5 second samples
```

The default output is to print byte summaries every five seconds.

Bytes. This example shows `iotop` run on Solaris/UFS with the `-C` option to *not* clear the screen but instead provide a scrolling output:

```
solaris# iotop -C
Tracing... Please wait.
2005 Jul 16 00:34:40,  load: 1.21,  disk_r:  12891 KB,  disk_w:   1087 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D            BYTES
    0      3      0 fsflush          cmdk0   102   4 W              512
    0      3      0 fsflush          cmdk0   102   0 W            11776
    0  27751  20320 tar              cmdk0   102  16 W            23040
    0      3      0 fsflush          cmdk0   102   0 R            73728
    0      0      0 sched            cmdk0   102   0 R           548864
    0      0      0 sched            cmdk0   102   0 W          1078272
    0  27751  20320 tar              cmdk0   102  16 R          1514496
    0  27751  20320 tar              cmdk0   102   3 R         11767808
[...]
```

In the previous output, we can see that a `tar(1)` command is reading from the `cmdk0` disk, from several different slices (different minor numbers), on the last report focusing on 102,5 (an `ls -lL` in `/dev/dsk` can explain the number to slice mappings).

The `disk_r` and `disk_w` values give a summary of the overall activity in bytes.

Bytes can be used as a yardstick to determine which process is keeping the disks busy, but either of the delta times available from iotop would be more accurate (because they take into account whether the activity is random or sequential).

Disk Time. Here's an example of printing disk time using `-o`:

```
solaris# iotop -Co
Tracing... Please wait.
2005 Jul 16 00:39:03,  load: 1.10,  disk_r:   5302 KB,  disk_w:     20 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D         DISKTIME
```

continues

```
    0      0      0 sched            cmdk0   102    0 W             532
    0      0      0 sched            cmdk0   102    0 R          245398
    0  27758  20320 find             cmdk0   102    0 R         3094794

2005 Jul 16 00:39:08,  load: 1.14,  disk_r:   5268 KB,  disk_w:    273 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D         DISKTIME
    0      3      0 fsflush          cmdk0   102    0 W            2834
    0      0      0 sched            cmdk0   102    0 W          263527
    0      0      0 sched            cmdk0   102    0 R          285015
    0      3      0 fsflush          cmdk0   102    0 R          519187
    0  27758  20320 find             cmdk0   102    0 R         2429232
[...]
```

The disk time is given in microseconds. In the first sample, we can see the `find(1)` command caused a total of 3.094 seconds of disk time: The duration of the samples here is five seconds (the default), so it would be fair to say that the `find` command is keeping this single disk 60 percent busy.

Find vs. Bart. Solaris 10 introduced the `bart(1M)` command for gathering and comparing file checksums as a security fingerprinting tool. Bart will read a file sequentially so that its checksums can be calculated. It can be run with the `find` command to determine which files to checksum. Here the `-P` option is used to print disk time percentages as both execute:

```
solaris# iotop -PC 1
Tracing... Please wait.
2005 Nov 18 15:26:14,  load: 0.24,  disk_r:  13176 KB,  disk_w:      0 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D   %I/O
    0   2215   1663 bart             cmdk0   102    0 R     85

2005 Nov 18 15:26:15,  load: 0.25,  disk_r:   5263 KB,  disk_w:      0 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D   %I/O
    0   2214   1663 find             cmdk0   102    0 R     15
    0   2215   1663 bart             cmdk0   102    0 R     67

2005 Nov 18 15:26:16,  load: 0.25,  disk_r:   8724 KB,  disk_w:      0 KB

  UID    PID   PPID CMD              DEVICE  MAJ MIN D   %I/O
    0   2214   1663 find             cmdk0   102    0 R     10
    0   2215   1663 bart             cmdk0   102    0 R     71
[...]
```

In the previous output, `bart` and `find` jostle for disk access as they create a database of file checksums. The command was as follows:

```
# find / | bart create -I > /dev/null
```

Note that the `%I/O` is in terms of one disk. A %I/O of 200 would mean that two disks were effectively at 100 percent, or 4 disks at 50 percent, and so on. The percentages can be presented in a different way (percentage of total disks or max percentage of busiest disk) by editing the script.

iopattern

Another presentation of statistics is to print single-line summaries. The `iopattern` tool is a good example of this,[11] which provides top-level data as a starting point for further investigation.

Script

Much of the data summarized is straight from the io provider; what makes this script interesting is a measure of percent random vs. sequential I/O. It does this with the following code:

```
io:genunix::done
/self->ok/
{
      /*
       * Save details
       */
      this->loc = args[0]->b_blkno * 512;
      this->pre = last_loc[args[1]->dev_statname];
      diskr += args[0]->b_flags & B_READ ? args[0]->b_bcount : 0;
      diskw += args[0]->b_flags & B_READ ? 0 : args[0]->b_bcount;
      diskran += this->pre == this->loc ? 0 : 1;
      diskcnt++;
      diskmin = diskmin == 0 ? args[0]->b_bcount :
          (diskmin > args[0]->b_bcount ? args[0]->b_bcount : diskmin);
      diskmax = diskmax < args[0]->b_bcount ? args[0]->b_bcount : diskmax;

      /* save disk location */
      last_loc[args[1]->dev_statname] = this->loc + args[0]->b_bcount;

      /* cleanup */
      self->ok = 0;
}
```

A `diskran` variable counts disk seeks for the percent random calculation and is incremented if the location of the current disk I/O is not equal to where the disk heads were after the last I/O. Although this works, it doesn't take into account the size of the seek—a workload is random whether it seeked over a short or long range. If this simplification becomes problematic, the script could be adjusted to express percent random I/O in terms of time spent serving random I/O vs. sequential I/O, rather than counts.

11. Thanks to Ryan Matteson for the idea.

The full script can be found in the DTraceToolkit and /usr/bin/iopattern on Mac OS X.

Examples

Here we show using the iopattern script while tracing known disk I/O loads.

Usage. The usage of iopattern can be printed with -h:

```
# iopattern -h
USAGE: iopattern [-v] [-d device] [-f filename] [-m mount_point]
                 [interval [count]]

                -v              # print timestamp
                -d device       # instance name to snoop
                -f filename     # snoop this file only
                -m mount_point  # this FS only
   eg,
        iopattern         # default output, 1 second samples
        iopattern 10      # 10 second samples
        iopattern 5 12    # print 12 x 5 second samples
        iopattern -m /    # snoop events on file system / only
```

Unlike similar one-line summary tools (such as vmstat(1M)), there is no "summary since boot" line printed (because DTrace wasn't running to trace the activity).

Sequential I/O. To demonstrate sequential I/O, a dd(1) command was executed on a raw disk device path to intentionally create sequential disk activity:

```
solaris# iopattern
%RAN %SEQ  COUNT    MIN    MAX    AVG     KR     KW
   1   99    465   4096  57344  52992  23916    148
   0  100    556  57344  57344  57344  31136      0
   0  100    634  57344  57344  57344  35504      0
   6   94    554    512  57344  54034  29184     49
   0  100    489  57344  57344  57344  27384      0
  21   79    568   4096  57344  46188  25576     44
   4   96    431   4096  57344  56118  23620      0
^C
```

In the previous output, the high percentages in the %SEQ column indicate that disk activity is mostly sequential. The disks are also pulling around 30MB during each sample, with a large average event size.

Random I/O. Here a find(1) command was used to cause random disk activity:

```
solaris# iopattern
%RAN %SEQ  COUNT    MIN    MAX    AVG     KR     KW
  86   14    400   1024   8192   1543    603      0
  81   19    455   1024   8192   1606    714      0
  89   11    469    512   8192   1854    550    299
  83   17    463   1024   8192   1782    806      0
  87   13    394   1024   8192   1551    597      0
  85   15    348    512  57344   2835    808    155
  91    9    513    512  47616   2812    570    839
  76   24    317    512  35840   3755    562    600
^C
```

In the previous output, we can see from the percentages that the disk events were mostly random. We can also see that the average event size is small, which makes sense if we are reading through many directory files.

geomiosnoop.d

All the previous scripts use the io provider and execute on Solaris, OpenSolaris, and Mac OS X. Until FreeBSD supports the io provider as well, the fbt provider can be used to retrieve similar information from the kernel. This script demonstrates this by snooping I/O requests to the `GEOM(4)` I/O framework. Since it uses the fbt provider, it will require maintenance to match changes in the FreeBSD kernel.

Script

To see both VFS and device I/O, two different GEOM functions are traced:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

/* from /usr/src/sys/sys/bio.h */
inline int BIO_READ = 0x01;
inline int BIO_WRITE = 0x02;

dtrace:::BEGIN
{
      printf("%5s %5s %1s %10s %6s %16s %-8s %s\n", "UID", "PID", "D",
           "OFFSET(KB)", "BYTES", "COMM", "VNODE", "INFO");
}

fbt::g_vfs_strategy:entry
{
      /* attempt to fetch the filename from the namecache */
      this->file = args[1]->b_vp->v_cache_dd != NULL ?
          stringof(args[1]->b_vp->v_cache_dd->nc_name) : "<unknown>";
      printf("%5d %5d %1s %10d %6d %16s %-8x %s \n", uid, pid,
          args[1]->b_iocmd & BIO_READ ? "R" : "W",
          args[1]->b_iooffset / 1024, args[1]->b_bcount,
          execname, (uint64_t)args[1]->b_vp, this->file);
}
```

continues

```
26
27    fbt::g_dev_strategy:entry
28    {
29          printf("%5d %5d %1s %10d %6d %16s %-8s %s\n", uid, pid,
30              args[0]->bio_cmd & BIO_READ ? "R" : "W",
31              args[0]->bio_offset / 1024, args[0]->bio_bcount,
32              execname, "<dev>", stringof(args[0]->bio_dev->si_name));
33    }
```

Script geomiosnoop.d

This traces GEOM requests via `g_vfs_strategy()` and `g_dev_strategy()` so that the process name that requested the I/O can be traced. GEOM may transform these requests into multiple other I/Os, which are later issued to the device via `g_down`.

Lines 19 and 20 attempt to fetch the filename from the `namecache` entry of the vnode, which is unreliable (when not cached). To compensate a little, the address of the vnode is printed in the `VNODE` column: This will at least help identify whether the same file or different files are being accessed. A more reliable way of determining the path name from a vnode is demonstrated in `vfssnoop.d` in Chapter 5, File Systems.

Example

Disk I/O requests from a couple of commands were caught; first, the `dd(1)` command was used to perform five 1KB reads from `/dev/ad0`, which was traced correctly. Then the `bsdtar(1)` command was used to archive the `/usr` file system, for which some of the filenames were identified and some not (either not related to a particular file or not in the name cache).

```
freebsd# geomiosnoop.d
  UID   PID D OFFSET(KB)  BYTES            COMM VNODE    INFO
    0 38004 R          0   1024              dd <dev>    ad0
    0 38004 R          1   1024              dd <dev>    ad0
    0 38004 R          2   1024              dd <dev>    ad0
    0 38004 R          3   1024              dd <dev>    ad0
    0 38004 R          4   1024              dd <dev>    ad0
    0 38005 R   11754552   2048          bsdtar c3f4eb84 etalon
    0 38005 R   11754554   2048          bsdtar c4427754 <unknown>
    0 38005 R   11727278   2048          bsdtar c3dd6b84 geom_stripe
    0 38005 R   11754556   2048          bsdtar c442796c <unknown>
    0 38005 R   11754558   2048          bsdtar c4427000 <unknown>
    0 38005 R   11747062   2048          bsdtar c4427d9c <unknown>
    0 38005 R   11727276   2048          bsdtar c3dd6c90 geom_shsec
    0 38005 R   11747064   2048          bsdtar c4427c90 <unknown>
    0 38005 R   11747066   2048          bsdtar c4427860 <unknown>
    0 38005 R   11747068   2048          bsdtar c4427b84 <unknown>
    0 38005 R   11727302   2048          bsdtar c3cbf53c lib
    0 38005 R   11743946   2048          bsdtar c442710c msun
[...]
```

SCSI Scripts

These scripts use the fbt provider to trace the SCSI driver(s). Small Computer System Interface (SCSI) is a commonly used interface for managing disk devices, especially for external storage disks. DTracing the SCSI driver can provide details of disk I/O operation at a lower level than with the io provider alone. Functionally, it can be summarized as in Figure 4-5.

The high-level diagram is in the "Capabilities" section. Note that both the physical and multipathing layers may work with SCSI I/O, so a single disk I/O request may be processed by multiple SCSI I/O events.

Since there is currently no stable SCSI provider, the fbt[12] and sdt providers are used. These are unstable interfaces: They expose kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for SCSI analysis.

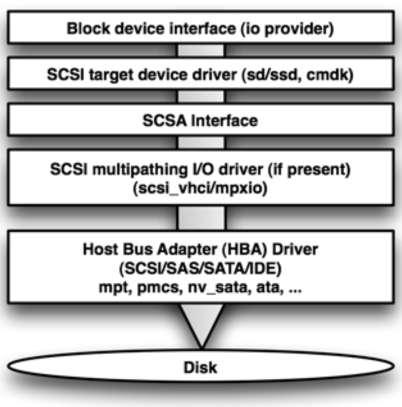


Figure 4-5 SCSI I/O stack

12. See the "fbt Provider" section in Chapter 12 for more discussion about the use of the fbt provider.

The SCSI Probes

To become familiar with the probes available for DTracing SCSI events, a disk I/O workload was applied while probes from the sdt and fbt providers were frequency counted. Starting with sdt for the sd module (no sdt probes currently exist in the `scsi` module itself):

```
# dtrace -n 'sdt:sd:: { @[probename] = count(); }'
dtrace: description 'sdt:sd:: ' matched 1 probe
^C

  scsi-transport-dispatch                                        3261
```

This has matched a single probe, `scsi-transport-dispatch`. This probe can be used to examine I/O dispatched from the sd target driver.

For the fbt provider, all probes from the `sd` and `scsi` kernel modules were frequency counted:

```
solaris# dtrace -n 'fbt:scsi::entry,fbt:sd::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:scsi::entry,fbt:sd::entry ' matched 585 probes
^C

  sd_pm_idletimeout_handler                                        40
  ddi_xbuf_qstrategy                                             3135
  scsi_hba_pkt_alloc                                             3135
  scsi_pkt_size                                                  3135
  sd_add_buf_to_waitq                                            3135
  sd_core_iostart                                                3135
  sd_initpkt_for_buf                                             3135
  sd_mapblockaddr_iostart                                        3135
  sd_setup_rw_pkt                                                3135
  sd_xbuf_init                                                   3135
  sd_xbuf_strategy                                               3135
  sdinfo                                                         3135
  sdstrategy                                                     3135
  xbuf_iostart                                                   3135
  scsi_hba_pkt_comp                                              3140
  ddi_xbuf_done                                                  3141
  ddi_xbuf_get                                                   3141
  scsi_hba_pkt_free                                              3141
  sd_buf_iodone                                                  3141
  sd_destroypkt_for_buf                                          3141
  sd_mapblockaddr_iodone                                         3141
  sd_return_command                                              3141
  sdintr                                                         3141
  xbuf_dispatch                                                  3141
  scsi_init_pkt                                                  6270
  scsi_device_hba_private_get                                    6271
  scsi_transport                                                 6272
  sd_start_cmds                                                  6276
  scsi_destroy_pkt                                               6282
  scsi_pkt_allocated_correctly                                   9416
  scsi_address_device                                           12542
```

As will be shown in the scripts that follow, two useful functions to probe are `scsi_transport()` and `scsi_destroy_pkt()`, to examine the start and end of all SCSI commands.

sdqueue.d

Although disk I/O latency was measured in the previous io provider scripts, another source of latency was not: the sd queue. Here is an example for those familiar with the output of `iostat(1M)` on Solaris:

```
# iostat -xnz 10
[...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  235.2    0.0 1882.0    0.0 94.0 34.0  399.5  144.5 100 100 c4t5000C50010743CFAd0
```

The previous io provider scripts gave us extended visibility into the `asvc_t` metric (active service time) but not `wsvc_t` (wait service time) due to queueing in sd. The `sdqueue.d` script traces the time I/O spends queued in sd and shows this by device as a distribution plot.

Script

This script uses an fbt probe to trace when I/Os are added to an sd queue, and it uses an `sdt` probe to trace when they are removed from the queue and dispatched. It was written for Solaris Nevada circa December 2009 and does not work on other Solaris versions that do not have the `sd_add_buf_to_waitq()` kernel function; to get this to work on those versions, try to find a similar function that is available and modify the script to use it.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::sd_add_buf_to_waitq:entry
/args[1]->b_dip/
{
         start_time[arg1] = timestamp;
}

sdt:::scsi-transport-dispatch
/this->start = start_time[arg0]/
{
         this->delta = (timestamp - this->start) / 1000;
```

continues

```
20          this->bp = (buf_t *)arg0;
21          this->dev = xlate <devinfo_t *>(this->bp)->dev_statname;
22          this->path = xlate <devinfo_t *>(this->bp)->dev_pathname;
23          @avg[this->dev, this->path] = avg(this->delta);
24          @plot[this->dev, this->path] = lquantize(this->delta / 1000, 0, 1000,
25              100);
26          start_time[arg0] = 0;
27  }
28
29  dtrace:::END
30  {
31          printf("Wait queue time by disk (ms):\n");
32          printa("\n  %-12s %-50s\n%@d", @plot);
33          printf("\n\n  %-12s %-50s %12s\n", "DEVICE", "PATH", "AVG_WAIT(us)");
34          printa("  %-12s %-50s %@12d\n", @avg);
35  }
```

Script sdqueue.d

Translators from the io provider (`/usr/lib/dtrace/io.d`) were used to convert a `buf_t` into the device name and path, on lines 21 and 22. These translators generated some errors when NFS I/O was traced, which is not the point of this script, and has been filtered out on line 11 by ensuring that this is for a local device.

Example

A Solaris system has 128 application threads calling random disk I/O on the same disk. The disk is saturated with I/O, and queue time dominates the latency—as shown by the `iostat(1M)` output earlier. Here is the output of `sdqueue.d`:

```
solaris# sdqueue.d
Tracing... Hit Ctrl-C to end.
^C
Wait queue time by disk (ms):

  sd116        /devices/scsi_vhci/disk@g5000c50010743cfa:wd

           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@                                    303
             100 |@@@@@                                    258
             200 |@@@@@                                    301
             300 |@@@@@                                    280
             400 |@@@@@                                    271
             500 |@@@@@                                    271
             600 |@@@@@                                    264
             700 |@@@@                                     205
             800 |@                                        59
             900 |                                         1
         >= 1000 |                                         0


  DEVICE       PATH                                               AVG_WAIT(us)
  sd116        /devices/scsi_vhci/disk@g5000c50010743cfa:wd             394465
```

The average wait time corresponds to the wsvc_t shown in the earlier iostat(1M) output. Here we can see the distribution plot of wait time that caused that average. This shows that 60 I/Os waited for longer than 800 ms—information that is lost when we look only at averages. Although 800 ms queue time sounds alarming, reconsider the workload: 128 application threads pounding on a single disk. All this I/O can't be delivered to the disk for its own on-disk queue, because that has a limited length. The overflow queues in sd and waits there, as measured by sdqueue.d.

sdretry.d

The SCSI disk driver can retry disk I/O if a problem was encountered. If this begins to happen frequently on a disk, it may mean that the disk is deteriorating and might fail soon. Surprisingly, this can occur in the Solaris SCSI driver without incrementing any statistics (such as soft errors in iostat(1M)). Although these I/Os do eventually complete successfully, it would be useful for us to know that they encountered problems on the disk and needed to retry. This can be observed using DTrace.

The fbt provider is used to trace the following function from uts/common/io/scsi/targets/sd.c:

```
/*
 *    Function: sd_set_retry_bp
 *
 * Description: Set up the given bp for retry.
 *
 *   Arguments: un - ptr to associated softstate
 *              bp - ptr to buf(9S) for the command
 *              retry_delay - time interval before issuing retry (may be 0)
 *              statp - optional pointer to kstat function
 *
 *     Context: May be called under interrupt context
 */

static void
sd_set_retry_bp(struct sd_lun *un, struct buf *bp, clock_t retry_delay,
        void (*statp)(kstat_io_t *))
{
```

Because this is a kernel function, there is no guarantee that this will not change in the very next release of OpenSolaris, breaking the sdretry.d script. That's a drawback inherent with fbt. Should sdretry.d become a popular script and its breakage become a nuisance, a static probe could be added to the kernel to ensure a stable interface.

Script

This script was written for Solaris Nevada circa December 2009; it may work on other Solaris versions, depending on how much the `sd_set_retry_bp()` function has changed.

Although this is an unstable fbt script, it fortunately uses only one fbt probe, so maintenance as the underlying kernel changes may not be too difficult. This script uses the existing io provider translator via `xlate` to fetch the device name and major and minor numbers from `args[1]`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... output every 10 seconds.\n");
}

fbt::sd_set_retry_bp:entry
{
         @[xlate <devinfo_t *>(args[1])->dev_statname,
             xlate <devinfo_t *>(args[1])->dev_major,
             xlate <devinfo_t *>(args[1])->dev_minor] = count();
}

tick-10sec
{
         printf("\n%Y:\n", walltimestamp);
         printf("%28s  %-3s,%-4s  %s\n", "DEVICE", "MAJ", "MIN", "RETRIES");
         printa("%28s  %-03d,%-4d  %@d\n", @);
         trunc(@);
}
```

Script sdretry.d

Examples

The following examples demonstrate the use of `sdretry.d`.

Unexpected Retries. This was executed on a system with 24 disks performing I/O, plus two (mostly) idle system disks. This is a system running normally with no errors visible in `iostat(1M)`:

```
solaris# iostat -e
           ---- errors ---
device  s/w h/w trn tot
sd6       0   0   0   0
sd7       0   0   0   0
sd107     0   0   0   0
sd108     0   0   0   0
sd109     0   0   0   0
sd110     0   0   0   0
```

```
sd111       0    0    0    0
sd112       0    0    0    0
sd113       0    0    0    0
sd114       0    0    0    0
sd115       0    0    0    0
sd116       0    0    0    0
sd117       0    0    0    0
sd118       0    0    0    0
sd119       0    0    0    0
sd120       0    0    0    0
sd121       0    0    0    0
sd122       0    0    0    0
sd123       0    0    0    0
sd124       0    0    0    0
sd125       0    0    0    0
sd126       0    0    0    0
sd127       0    0    0    0
sd128       0    0    0    0
sd129       0    0    0    0
sd130       0    0    0    0
```

For a system with no apparent disk issues, we would expect no, or very few, SCSI retries. However, running the `sdretry.d` script immediately caught retries:

```
solaris# sdretry.d
Tracing... output every 10 seconds.

2009 Dec 28 04:32:19:
                            DEVICE  MAJ,MIN   RETRIES
                               sd7  227,448   15
                               sd6  227,384   17

2009 Dec 28 04:32:29:
                            DEVICE  MAJ,MIN   RETRIES

2009 Dec 28 04:32:39:
                            DEVICE  MAJ,MIN   RETRIES
                               sd6  227,384   4
                               sd7  227,448   7

2009 Dec 28 04:32:49:
                            DEVICE  MAJ,MIN   RETRIES
                               sd6  227,384   9
                               sd7  227,448   16
```

We see retries not on the storage disks but on the system disks (sd6 and sd7). These were mostly idle, flushing monitoring data to disk every five seconds or so—and, as it turns out, encountering SCSI retries.

Known Disk Issue. Here a disk was removed during disk I/O[13] and reinserted a few seconds later:

13. This is not recommended.

```
# sdretry.d
2009 Dec 28 05:05:55:
                      DEVICE  MAJ,MIN   RETRIES
                       sd123  227,7872  6
```

`sdretry.d` has picked up the disk which was pulled, which experienced six retries. Interestingly, these were not counted as any type of error:

```
# iostat -e
           ---- errors ---
device  s/w h/w trn tot
[...]
sd123     0   0   0   0
```

scsicmds.d

Apart from read and write I/O, there are many other SCSI commands that can be sent to process I/O and manage SCSI devices. The `scsicmds.d` script frequency counts these SCSI commands by type.

Script

Most of this script is the `dtrace:::BEGIN` statement, which defines an associative array for translating a SCSI command code into a human-readable string. This block of code was autogenerated by processing a SCSI definitions header file.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
string scsi_cmd[uchar_t];
dtrace:::BEGIN
{
       /*
        * The following was generated from the SCSI_CMDS_KEY_STRINGS
        * definitions in /usr/include/sys/scsi/generic/commands.h using sed.
        */
        scsi_cmd[0x00] = "test_unit_ready";
        scsi_cmd[0x01] = "rezero/rewind";
        scsi_cmd[0x03] = "request_sense";
        scsi_cmd[0x04] = "format";
        scsi_cmd[0x05] = "read_block_limits";
        scsi_cmd[0x07] = "reassign";
        scsi_cmd[0x08] = "read";
        scsi_cmd[0x0a] = "write";
        scsi_cmd[0x0b] = "seek";
        scsi_cmd[0x0f] = "read_reverse";
        scsi_cmd[0x10] = "write_file_mark";
        scsi_cmd[0x11] = "space";
        scsi_cmd[0x12] = "inquiry";
        scsi_cmd[0x13] = "verify";
        scsi_cmd[0x14] = "recover_buffer_data";
        scsi_cmd[0x15] = "mode_select";
        scsi_cmd[0x16] = "reserve";
        scsi_cmd[0x17] = "release";
```

```
        scsi_cmd[0x18] = "copy";
        scsi_cmd[0x19] = "erase_tape";
        scsi_cmd[0x1a] = "mode_sense";
        scsi_cmd[0x1b] = "load/start/stop";
        scsi_cmd[0x1c] = "get_diagnostic_results";
        scsi_cmd[0x1d] = "send_diagnostic_command";
        scsi_cmd[0x1e] = "door_lock";
        scsi_cmd[0x23] = "read_format_capacity";
        scsi_cmd[0x25] = "read_capacity";
        scsi_cmd[0x28] = "read(10)";
        scsi_cmd[0x2a] = "write(10)";
        scsi_cmd[0x2b] = "seek(10)";
        scsi_cmd[0x2e] = "write_verify";
        scsi_cmd[0x2f] = "verify(10)";
        scsi_cmd[0x30] = "search_data_high";
        scsi_cmd[0x31] = "search_data_equal";
        scsi_cmd[0x32] = "search_data_low";
        scsi_cmd[0x33] = "set_limits";
        scsi_cmd[0x34] = "read_position";
        scsi_cmd[0x35] = "synchronize_cache";
        scsi_cmd[0x37] = "read_defect_data";
        scsi_cmd[0x39] = "compare";
        scsi_cmd[0x3a] = "copy_verify";
        scsi_cmd[0x3b] = "write_buffer";
        scsi_cmd[0x3c] = "read_buffer";
        scsi_cmd[0x3e] = "read_long";
        scsi_cmd[0x3f] = "write_long";
        scsi_cmd[0x44] = "report_densities/read_header";
        scsi_cmd[0x4c] = "log_select";
        scsi_cmd[0x4d] = "log_sense";
        scsi_cmd[0x55] = "mode_select(10)";
        scsi_cmd[0x56] = "reserve(10)";
        scsi_cmd[0x57] = "release(10)";
        scsi_cmd[0x5a] = "mode_sense(10)";
        scsi_cmd[0x5e] = "persistent_reserve_in";
        scsi_cmd[0x5f] = "persistent_reserve_out";
        scsi_cmd[0x80] = "write_file_mark(16)";
        scsi_cmd[0x81] = "read_reverse(16)";
        scsi_cmd[0x83] = "extended_copy";
        scsi_cmd[0x88] = "read(16)";
        scsi_cmd[0x8a] = "write(16)";
        scsi_cmd[0x8c] = "read_attribute";
        scsi_cmd[0x8d] = "write_attribute";
        scsi_cmd[0x8f] = "verify(16)";
        scsi_cmd[0x91] = "space(16)";
        scsi_cmd[0x92] = "locate(16)";
        scsi_cmd[0x9e] = "service_action_in(16)";
        scsi_cmd[0x9f] = "service_action_out(16)";
        scsi_cmd[0xa0] = "report_luns";
        scsi_cmd[0xa2] = "security_protocol_in";
        scsi_cmd[0xa3] = "maintenance_in";
        scsi_cmd[0xa4] = "maintenance_out";
        scsi_cmd[0xa8] = "read(12)";
        scsi_cmd[0xa9] = "service_action_out(12)";
        scsi_cmd[0xaa] = "write(12)";
        scsi_cmd[0xab] = "service_action_in(12)";
        scsi_cmd[0xac] = "get_performance";
        scsi_cmd[0xAF] = "verify(12)";
        scsi_cmd[0xb5] = "security_protocol_out";
        printf("Tracing... Hit Ctrl-C to end.\n");
}
fbt::scsi_transport:entry
{
```

continues

```
91      this->dev = (struct dev_info *)args[0]->pkt_address.a_hba_tran->tran_hba_dip;
92          this->nodename = this->dev != NULL ?
93              stringof(this->dev->devi_node_name) : "<unknown>";
94          this->code = *args[0]->pkt_cdbp;
95          this->cmd = scsi_cmd[this->code] != NULL ?
96              scsi_cmd[this->code] : lltostr(this->code);
97          @[this->nodename, this->cmd] = count();
98  }
99  dtrace:::END
100 {
101          printf("  %-24s %-36s  %s\n", "DEVICE NODE", "SCSI COMMAND", "COUNT");
102          printa("  %-24s %-36s  %@d\n", @);
103 }
```

scsicmds.d

Apart from aggregating on SCSI I/O type, the script also prints the `DEVICE NODE` to indicate which layer (see Figure 4-5) those SCSI commands apply to. Fetching the device node information involves walking several kernel structures, which is the part of the script mostly likely to break (and require modification) in future kernel updates.

Note the use of `lltostr()` on line 96: If a SCSI command cannot be found in the translation array, its numerical code is converted to a string using `lltostr()`, which is used in lieu of the missing string description.

Examples

Examples here include read workload and zpool status.

Read Workload. A read-intensive workload was occurring on a Solaris system with 24 external storage disks configured with multipathing. The system also has two internal system disks (no multipathing), which are occasionally writing monitoring data.

```
solaris# scsicmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE               SCSI COMMAND                          COUNT
  pci1000,3150              inquiry                               4
  scsi_vhci                 inquiry                               4
  pci10de,cb84              write                                 8
  pci10de,cb84              synchronize_cache                     22
  pci1000,3150              synchronize_cache                     50
  scsi_vhci                 synchronize_cache                     50
  pci1000,3150              get_diagnostic_results                117
  scsi_vhci                 get_diagnostic_results                117
  pci1000,3150              write(10)                             203
  scsi_vhci                 write(10)                             203
  pci10de,cb84              write(10)                             658
  pci1000,3150              read(10)                              535701
  scsi_vhci                 read(10)                              535701
```

The SCSI writes to the system disks can be identified under the device node pci10de,cb84. The remaining SCSI commands appear on both the multipathing device scsi_vhci and the physical device paths to the external storage disks pci1000,3150. The kernel module names for these device node names can be seen in /etc/path_to_inst (for example, scsi_vhci devices are handled by the SCSI disk driver sd, and pci1000,3150 is handled by the SAS HBA driver mpt); or this DTrace script can be enhanced to dig out those module names as well.

zpool Status. ZFS has a zpool status command, which lists the status of all the pool disks. It fetches disk status using SCSI commands, which can be seen using scsicmds.d:

```
solaris# scsicmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE              SCSI COMMAND                          COUNT
  pci10de,cb84             mode_sense                            4
  pci10de,cb84             test_unit_ready                       8
  pci10de,cb84             read                                  12
  pci1000,3150             mode_sense                            88
  pci1000,3150             test_unit_ready                       88
  scsi_vhci                mode_sense                            88
  scsi_vhci                test_unit_ready                       88
  pci1000,3150             read                                  132
  scsi_vhci                read                                  132
```

If particular commands are of interest, the script could be customized to match on them and dig out more information from the available function arguments.

scsilatency.d

The disklatency.d script earlier showed I/O latency in terms of disk; the scsilatency.d script shows I/O latency in terms of SCSI command and completion reason.

Script

This script also uses an associative array to convert SCSI commands into text descriptions, however, only 16 of the most common commands are included. This may be sufficient and reduces the length of the script. Latency is measured as the time from scsi_transport() to scsi_destroy_pkt():

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

```

continues

```
string scsi_cmd[uchar_t];

dtrace:::BEGIN
{
        /* See /usr/include/sys/scsi/generic/commands.h for the full list. */
        scsi_cmd[0x00] = "test_unit_ready";
        scsi_cmd[0x08] = "read";
        scsi_cmd[0x0a] = "write";
        scsi_cmd[0x12] = "inquiry";
        scsi_cmd[0x17] = "release";
        scsi_cmd[0x1a] = "mode_sense";
        scsi_cmd[0x1b] = "load/start/stop";
        scsi_cmd[0x1c] = "get_diagnostic_results";
        scsi_cmd[0x1d] = "send_diagnostic_command";
        scsi_cmd[0x25] = "read_capacity";
        scsi_cmd[0x28] = "read(10)";
        scsi_cmd[0x2a] = "write(10)";
        scsi_cmd[0x35] = "synchronize_cache";
        scsi_cmd[0x4d] = "log_sense";
        scsi_cmd[0x5e] = "persistent_reserve_in";
        scsi_cmd[0xa0] = "report_luns";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::scsi_transport:entry
{
        start[arg0] = timestamp;
}

fbt::scsi_destroy_pkt:entry
/start[arg0]/
{
        this->delta = (timestamp - start[arg0]) / 1000;
        this->code = *args[0]->pkt_cdbp;
        this->cmd = scsi_cmd[this->code] != NULL ?
            scsi_cmd[this->code] : lltostr(this->code);
        this->reason = args[0]->pkt_reason == 0 ? "Success" :
            strjoin("Fail:", lltostr(args[0]->pkt_reason));

        @num[this->cmd, this->reason] = count();
        @average[this->cmd, this->reason] = avg(this->delta);
        @total[this->cmd, this->reason] = sum(this->delta);

        start[arg0] = 0;
}

dtrace:::END
{
        normalize(@total, 1000);
        printf("\n  %-26s %-12s %11s %11s %11s\n", "SCSI COMMAND",
            "COMPLETION", "COUNT", "AVG(us)", "TOTAL(ms)");
        printa("  %-26s %-12s %@11d %@11d %@11d\n", @num, @average, @total);
}
```

Script scsilatency.d

SCSI command failures are detected by checking `pkt_reason`, which will be zero for success and some other code for failure (translated in the `scsireasons.d` script that follows). For failure, the script will include the failed reason code for reference.

Example

On a server with an external storage array performing a heavy disk read workload, a disk was removed to cause some I/O to fail:

```
solaris# scsilatency.d
Tracing... Hit Ctrl-C to end.
^C

  SCSI COMMAND                COMPLETION          COUNT      AVG(us)    TOTAL(ms)
  release                     Success                 2          128            0
  persistent_reserve_in       Success                 2          161            0
  report_luns                 Success                 2       175346          350
  load/start/stop             Success                 6        10762           64
  send_diagnostic_command     Success                12         8785          105
  read(10)                    Fail:4                 12      1276114        15313
  write                       Success                16          479            7
  read_capacity               Success                48          154            7
  inquiry                     Success               150         3294          494
  read                        Success               158         5438          859
  log_sense                   Success               190        43011         8172
  test_unit_ready             Success               201         3529          709
  mode_sense                  Success               386        40300        15555
  synchronize_cache           Success               780        36601        28548
  get_diagnostic_results      Success              2070        10944        22654
  write(10)                   Success              6892        10117        69731
  read(10)                    Success           1166169        23516     27424549
```

The successful reads had an average latency of 24 ms. However, the failed reads had an average latency of more than 1.27 seconds! The reason for the long latency would be SCSI retries, timeouts, or a combination of both. (The previous `sdretry.d` script can be used to confirm that retries occurred.) This script also identified a SCSI command, which is particularly slow despite returning successfully: the `report luns` command, with an average time of 175 ms.

scsirw.d

This script takes the three common SCSI commands—`read`, `write`, and `sync-cache`—and prints summaries of the I/O sizes and times. This provides visibility for the I/O throughput rates and latencies for SCSI.

Script

Information about the I/O, including start time and size, is cached on the return of the `scsi_init_pkt()` function, since its return value (`arg1`) is the packet address and is used as a key for associative arrays. Command completion is traced using `scsi_destroy_pkt()`, which takes the packet address as the argument and uses it to look up the previous associative arrays.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::sd_setup_rw_pkt:entry { self->in__sd_setup_rw_pkt = 1; }
fbt::sd_setup_rw_pkt:return { self->in__sd_setup_rw_pkt = 0; }

fbt::scsi_init_pkt:entry
/self->in__sd_setup_rw_pkt/
{
        self->buf = args[2];
}

/* Store start time and size for read and write commands */
fbt::scsi_init_pkt:return
/self->buf/
{
        start[arg1] = timestamp;
        size[arg1] = self->buf->b_bcount;
        dir[arg1] = self->buf->b_flags & B_WRITE ? "write" : "read";
        self->buf = 0;
}

fbt::sd_send_scsi_SYNCHRONIZE_CACHE:entry { self->in__sync_cache = 1; }
fbt::sd_send_scsi_SYNCHRONIZE_CACHE:return { self->in__sync_cache = 0; }

/* Store start time for sync-cache commands */
fbt::scsi_init_pkt:return
/self->in__sync_cache/
{
        start[arg1] = timestamp;
        dir[arg1] = "sync-cache";
}

/* SCSI command completed */
fbt::scsi_destroy_pkt:entry
/start[arg0]/
{
        this->delta = (timestamp - start[arg0]) / 1000;
        this->size = size[arg0];
        this->dir = dir[arg0];

        @num[this->dir] = count();
        @avg_size[this->dir] = avg(this->size);
        @avg_time[this->dir] = avg(this->delta);
        @sum_size[this->dir] = sum(this->size);
        @sum_time[this->dir] = sum(this->delta);
        @plot_size[this->dir] = quantize(this->size);
        @plot_time[this->dir] = quantize(this->delta);

        start[arg0] = 0;
        size[arg0] = 0;
        dir[arg0] = 0;
}

dtrace:::END
{
        normalize(@avg_size, 1024);
        normalize(@sum_size, 1048576);
        normalize(@sum_time, 1000);
```

```
        printf("  %-10s  %10s  %10s %10s  %10s %12s\n", "DIR",
            "COUNT", "AVG(KB)", "TOTAL(MB)", "AVG(us)", "TOTAL(ms)");
        printa("  %-10s  %@10d  %@10d %@10d  %@10d %@12d\n", @num,
            @avg_size, @sum_size, @avg_time, @sum_time);
        printf("\n\nSCSI I/O size (bytes):\n");
        printa(@plot_size);
        printf("\nSCSI I/O latency (us):\n");
        printa(@plot_time);
}
```

Script diskIO_scsirw.d

Example

The following example traced read I/O and synchronous write I/O to a ZFS file system. ZFS is calling the `sync-cache` commands to ensure that the synchronous writes are properly sent to disk.

```
solaris# scsirw.d
Tracing... Hit Ctrl-C to end.
^C
  DIR                COUNT     AVG(KB)  TOTAL(MB)     AVG(us)    TOTAL(ms)
  sync-cache           490           0          0       11034         5407
  write                647           9          5         608          393
  read                1580           3          4        1258         1987


SCSI I/O size (bytes):

  sync-cache
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 490
               1 |                                         0

  read
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@                   877
            1024 |                                         5
            2048 |@@@@@@                                   240
            4096 |@@@@@@@@                                 333
            8192 |@@                                       94
           16384 |@                                        23
           32768 |                                         1
           65536 |                                         7
          131072 |                                         0

  write
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@                                       37
            1024 |@@@@                                     72
            2048 |@@@                                      48
            4096 |@                                        16
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            474
           16384 |                                         0


SCSI I/O latency (us):
```

continues

```
   write
            value  ------------- Distribution ------------- count
               64 |                                         0
              128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            467
              256 |                                         5
              512 |@@                                       33
             1024 |@@@@@                                    83
             2048 |@@@@                                     58
             4096 |                                         1
             8192 |                                         0

   read
            value  ------------- Distribution ------------- count
               32 |                                         0
               64 |@@@@@@@@@@@@@                            526
              128 |@@@@@@@@@@@@@@@@@@@@                     789
              256 |@                                        27
              512 |                                         12
             1024 |                                         16
             2048 |@                                        27
             4096 |@@                                       83
             8192 |@@                                       93
            16384 |                                         7
            32768 |                                         0

   sync-cache
            value  ------------- Distribution ------------- count
             1024 |                                         0
             2048 |@                                        14
             4096 |@@@@@@@                                  85
             8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             344
            16384 |@@@                                      39
            32768 |@                                        8
            65536 |                                         0
```

The latency strongly suggests that the disk has an onboard write cache enabled, since the write operations (which copy the data to the disk device) are often returning in the 128- to 155-us range, whereas the `sync-cache` commands are often returning in at least 8 ms. The fast write times are just showing the time to copy the data to disk; the actual time to write the data to the stable storage device is better reflected in the `sync-cache` value.

Here is a sample `iostat(1M)` output while the synchronous write workload continued to run:

```
  # iostat -xnz 1
  [...]
                      extended device statistics
      r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
      0.0   83.0    0.0  996.1  0.0  0.0    0.0    0.2   0   1 c3t0d0
      0.0   84.0    0.0 1008.1  0.0  0.0    0.0    0.2   0   1 c3t1d0
  [...]
```

Based on those numbers, we would not think the disk was the bottleneck. However, `iostat(1M)` is not taking `sync-cache` into account, where the real time is spent waiting on disk.

scsireasons.d

When SCSI events complete, a reason code is set to show why the completion was sent. When errors occur, this can explain the nature of the error. The `scsireasons.d` script shows a summary of all SCSI completion reason codes and those that errored, along with the disk device name.

Script

As with the `scsicmds.d` script, here an associative array is declared to translate SCSI reason codes into human-readable text:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /*
         * The following was generated from the CMD_* pkt_reason definitions
         * in /usr/include/sys/scsi/scsi_pkt.h using sed.
         */
        scsi_reason[0] = "no transport errors- normal completion";
        scsi_reason[1] = "transport stopped with not normal state";
        scsi_reason[2] = "dma direction error occurred";
        scsi_reason[3] = "unspecified transport error";
        scsi_reason[4] = "Target completed hard reset sequence";
        scsi_reason[5] = "Command transport aborted on request";
        scsi_reason[6] = "Command timed out";
        scsi_reason[7] = "Data Overrun";
        scsi_reason[8] = "Command Overrun";
        scsi_reason[9] = "Status Overrun";
        scsi_reason[10] = "Message not Command Complete";
        scsi_reason[11] = "Target refused to go to Message Out phase";
        scsi_reason[12] = "Extended Identify message rejected";
        scsi_reason[13] = "Initiator Detected Error message rejected";
        scsi_reason[14] = "Abort message rejected";
        scsi_reason[15] = "Reject message rejected";
        scsi_reason[16] = "No Operation message rejected";
        scsi_reason[17] = "Message Parity Error message rejected";
        scsi_reason[18] = "Bus Device Reset message rejected";
        scsi_reason[19] = "Identify message rejected";
        scsi_reason[20] = "Unexpected Bus Free Phase occurred";
        scsi_reason[21] = "Target rejected our tag message";
        scsi_reason[22] = "Command transport terminated on request";
        scsi_reason[24] = "The device has been removed";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::scsi_init_pkt:entry
/args[2] != NULL/
{
        self->name = xlate <devinfo_t *>(args[2])->dev_statname;
}

fbt::scsi_init_pkt:return
{
        pkt_name[arg1] = self->name;
```

continues

```
        self->name = 0;
}

fbt::scsi_destroy_pkt:entry
{
        this->code = args[0]->pkt_reason;
        this->reason = scsi_reason[this->code] != NULL ?
            scsi_reason[this->code] : "<unknown reason code>";
        @all[this->reason] = count();
}

fbt::scsi_destroy_pkt:entry
/this->code != 0/
{
        this->name = pkt_name[arg0] != NULL ? pkt_name[arg0] : "<unknown>";
        @errors[pkt_name[arg0], this->reason] = count();
}

fbt::scsi_destroy_pkt:entry
{
        pkt_name[arg0] = 0;
}

dtrace:::END
{
        printf("\nSCSI I/O completion reason summary:\n");
        printa(@all);
        printf("\n\nSCSI I/O reason errors by disk device and reason:\n\n");
        printf("  %-16s  %-44s %s\n", "DEVICE", "ERROR REASON", "COUNT");
        printa("  %-16s  %-44s %@d\n", @errors);
}
```

Script scsireasons.d

The SCSI reason code isn't set properly until the command completes, so a completion event is traced: `scsi_destroy_pkt()`. We picked this because it happens during completion of SCSI I/O and has the `scsi_pkt` type as an argument. That's when things get difficult: The device details it can reference may have been cleared for failed devices, and we want to trace failed SCSI I/O using the device name. The device name is available when the command is issued, but this is a different thread, and the data can't be associated between the probes using a thread-local variable, `self->`.

To solve this, we trace `scsi_init_pkt()`, because it can access both the device info (on its entry argument) and a packet address (return argument). That packet address is used as a key to the `pkt_name` associative array, where the device name is cached. `scsi_destroy_pkt()` also has the packet address, which it can use as a key in that associative array to retrieve the device name.

This is one example solution for a script that shows SCSI return codes with device names, but it can be solved many other ways using DTrace.

Example

To trigger a SCSI error as an example, the `scsireasons.d` script was run while a disk was removed:

```
solaris# scsireasons.d
Tracing... Hit Ctrl-C to end.
^C

SCSI I/O completion reason summary:

  Target completed hard reset sequence                             12
  no transport errors- normal completion                       835258


SCSI I/O reason errors by disk device and reason:

  DEVICE            ERROR REASON                                    COUNT
  sd118             Target completed hard reset sequence            12
```

This has identified the problem disk as sd118, which had completed 12 hard resets.

scsi.d

`scsi.d` is a powerful DTrace script to trace or summarize SCSI I/O, showing details of the SCSI operations and latency times. It was written by Chris Gerhard[14] and Joel Buckley, and the latest version can be found at *http://blogs.sun.com/chrisg/page/scsi.d*. It has been written using the fbt provider so it is likely to work (without modification) on only some versions of the OpenSolaris kernel.

Script

`scsi.d` is almost 1,000 lines of DTrace—a little long to duplicate in this chapter (see the Web site for the full listing). It is written in a distinctive style, worth commenting both as an example of the DTrace scripting language as well the tracing of the SCSI driver; various parts of interest are shown here.

The first line

```
#!/usr/sbin/dtrace -qCs
```

shows that the `-C` option is used, causing DTrace to run the `cpp(1)` preprocessor and allowing the use of `#defines` and macros. Near the top of the script is a block comment to describes its usage:

14. He describes `scsi.d` in a blog post at *http://blogs.sun.com/chrisg/entry/scsi_d_script*.

```
/*
 * SCSI logging via dtrace.
 *
 * See http://blogs.sun.com/chrisg/tags/scsi.d
 *
 * Usage:
[...]
 *      -D EXECNAME='"foo"'
 * Which results scsi.d only reporting IO associated with the application "foo".
 *      -D PRINT_STACK
 * Which results in scsi.d printing a kernel stack trace after every outgoing
 * packet.
 *      -D QUIET
 * Which results in none of the packets being printed. Kind of pointless
 * without another option.
 *      -D PERF_REPORT
 * Which results in a report of how long IOs took aggregated per HBA useful
 * with -D QUIET to get performance statistics.
 * -D TARGET_STATS
 *      aggregate the stats based on the target.
 * -D LUN_STATS
 *      aggregate the stats based on the LUN. Requires TARGET_STATS
 * -D DYNVARSIZE
 *      pass this value to the #pragma D option dynvarsize= option.
 * -D HBA
 *      the name of the HBA we are interested in.
 * -D MIN_LBA
 *      Only report logical blocks over this value
 * -D MAX_LBA
 *      Only IOs to report logical blocks that are less than this value.
 * -D REPORT_OVERTIME=N
 *      Only report IOs that have taken longer than this number of nanoseconds.
 *      This only stops the printing of the packets not the collection of
 *      statistics.
 *      There are some tuning options that take effect only when
 *      REPORT_OVERTIME is set. These are:
 *      -D NSPEC=N
 *              Set the number of speculations to this value.
 *      -D SPECSIZE=N
 *              Set the size of the speculaton buffer.  This should be 200 *
 *              the size of NSPEC.
 *      -D CLEANRATE=N
 *              Specify the clean rate.
 *
 * Finally scsi.d will also now accept the dtrace -c and -p options to trace
 * just the commands or process given.
 *
 * Since dtrace does not output in real time it is useful to sort the output
 * of the script using sort -n to get the entries in chronological order.
 *
 * NOTE:  This does not directly trace what goes onto the scsi bus or fibre,
 * to do so would require this script have knowledge of every HBA that could
 * ever be connected to a system. It traces the SCSI packets as they are
 * passed from the target driver to the HBA in the SCSA layer and then back
 * again. Although to get the packet when it is returned it guesses that the
 * packet will be destroyed using scsi_destroy_pkt and not modified before it
 * is. So far this has worked but there is no garauntee that it will work for
 * all HBAs and target drivers in the future.
 *
 */
```

DTrace itself doesn't allow flexible programmable arguments such as with the `getopts()` function, so a workaround must be used to make DTrace script accept arguments. Earlier, with `iosnoop`, we demonstrated wrapping the DTrace script in a shell script; `scsi.d` makes use of the `-D` option to the `dtrace` command to define arguments.

The following may not look much like the other scripts is this book, but it's still DTrace:

```
226  #define P_TO_DEVINFO(pkt) ((struct dev_info *)(P_TO_TRAN(pkt)->tran_hba_dip))
227
228  #define DEV_NAME(pkt) \
229          stringof(`devnamesp[P_TO_DEVINFO(pkt)->devi_major].dn_name) /* ` */
230
231  #define DEV_INST(pkt) (P_TO_DEVINFO(pkt)->devi_instance)
232
233  #ifdef MIN_BLOCK
234  #define MIN_TEST && this->lba >= (MIN_BLOCK)
235  #else
236  #define MIN_TEST
237  #endif
```

The use of `-C` on line 1 allows the script to use such preprocessor syntax. The first three `#defines` reduce a lengthy statement into a short and readable macro that can be reused throughout the code. Also, note the odd comment at the end of line 229: `/* ` */`. This does nothing for DTrace but does prevent some syntax highlighting text editors getting their colors confused as they try to pair up backquotes with a certain color. There was another backquote on line 229, for the kernel symbol `` `devnamesp ``, which pairs with the one at the end of the line.

The following shows a translation associative array similar to the one in `scsicmds.d`:

```
332          scsi_ops[0x000, 0x0] = "TEST_UNIT_READY";
333          scsi_ops[0x001, 0x0] = "REZERO_UNIT_or_REWIND";
334          scsi_ops[0x003, 0x0] = "REQUEST_SENSE";
```

The names of the probes used are defined as macros `CDB_PROBES` and `ENTRY_PROBES`:

```
553  /* FRAMEWORK:- Add your probe name to the list for CDB_PROBES */
554  #define CDB_PROBES \
555  fbt:scsi:scsi_transport:entry, \
556  fbt:*sd:*sd_sense_key_illegal_request:entry
557
558  #define ENTRY_PROBES \
559  CDB_PROBES, \
560  fbt:scsi:scsi_destroy_pkt:entry
```

Since these are used in many action blocks in the script, we can tune their definition in one place by using a macro. To see how these macros are used, use this:

```
ENTRY_PROBES
/ this->arg_test_passed /
{
        SPECULATE
        PRINT_TIMESTAMP();
        PRINT_DEV_FROM_PKT(this->pkt);
        printf("%s 0x%2.2x %9s address %2.2d:%2.2d, lba 0x%*.*x, ",
[...]
```

The output line of text is generated with multiple print statements, with the last printing `\n`.

And here's something you don't see every day:

```
PRINT_CDB(0)
PRINT_CDB(1)
PRINT_CDB(2)
PRINT_CDB(3)
PRINT_CDB(4)
PRINT_CDB(5)
[...etc, to 31...]
```

`PRINT_CDB` is declared earlier to print the specified byte from the command block (`CDB`). Up to 32 bytes are printed; however, because the DTrace language does not support loops, this has been achieved by an *unrolled loop*.

Examples

Examples include disks reads and writes, disks reads with multipathing, and latency by driver instance.

Disks Reads and Writes. This shows the default output of `scsi.d` while a single read and then a single write I/O were issued. The disks were internal SATA disks accessed via the nv_sata driver:

```
solaris# scsi.d
00002.763975484 nv_sata4:-> 0x28  READ(10) address 00:00, lba 0x023e9082, len 0x000080
, control 0x00 timeout 5 CDBP ffffff8275b85d88 1 sched(0) cdb(10) 2800023e908200008000
00002.773631991 nv_sata4:<- 0x28  READ(10) address 00:00, lba 0x023e9082, len 0x000080
, control 0x00 timeout 5 CDBP ffffff8275b85d88, reason 0x0 (COMPLETED) pkt_state 0x1f
state 0x0 Success Time 9679us
00003.110393754 nv_sata4:-> 0x2a WRITE(10) address 00:00, lba 0x0474e714, len 0x000001
, control 0x00 timeout 5 CDBP ffffffda29746d88 1 sched(0) cdb(10) 2a000474e71400000100
00003.111017077 nv_sata4:<- 0x2a WRITE(10) address 00:00, lba 0x0474e714, len 0x000001
, control 0x00 timeout 5 CDBP ffffffda29746d88, reason 0x0 (COMPLETED) pkt_state 0x1f
state 0x0 Success Time 649us
```

Each SCSI event prints one output line, but these lines are more than 200 characters long, so the output has wrapped. The first two lines show a SCSI read command request and its return, followed by a SCSI write request and its return. As we can see, the time for the read was 9.7 ms, but the write occurred in 0.6 ms (perhaps because of disk write caching).

The fields of the default output are shown in Table 4-6.

Refer to documentation for the SCSI protocol for more details about these fields.

Table 4-6 scsi.d Output Fields

Number	Direction	Prefix	Field
1	<- ->		Elapsed time event occurred, in seconds
2	<- ->		Name of kernel driver calling a SCSI command
3	<- ->		Direction of SCSI command, either request -> or return <-
4	<- ->		SCSI command code, hexadecimal
5	<- ->		SCSI command code, text description
6	<- ->	`address`	SCSI address target:lun
7	<- ->	`lba`	Logical block address
8	<- ->	`len`	Length of I/O, hex number of sectors (1 sector == 512 bytes)
9	<- ->	`control`	SCSI command flags
10	<- ->	`timeout`	SCSI command timeout, seconds
11	<- ->	`CDBP`	Command block pointer (for use in `mdb -k`)
12	->		Process name (PID)
13	->	`cdb`	Command block length, bytes
14	->		Command block contents, hexadecimal
12	<-	`reason`	Command completion reason code, decimal
13	<-		Command completion reason code, text description
14	<-	`pkt_state`	State of SCSI command, number
15	<-	`state`	SCSI state, hexadecimal
16	<-		SCSI state, text description
17	<-	`Time`	SCSI command response time, microseconds

Disk Reads with Multipathing. This example shows a READ I/O to an external disk connected to the host system using multipathing:

```
solaris# scsi.d
[...]
00000.096386040 scsi_vhci0:-> 0x28  READ(10) address 2432:46, lba 0x0edd4e4e, le n 0x0
00100, control 0x00 timeout 5 CDBP ffffff89d1b44a08 1 sched(0) cdb(10) 2800 0edd4e4e00
010000
00000.096435411 mpt0:-> 0x28  READ(10) address 09:00, lba 0x0edd4e4e, len 0x0001 00, c
ontrol 0x00 timeout 5 CDBP ffffffc5e9f5e9a8 1 sched(0) cdb(10) 28000edd4e4e 00010000
[...]
00000.103215546 scsi_vhci0:<- 0x28  READ(10) address 2432:46, lba 0x0edd4e4e, le n 0x0
00100, control 0x00 timeout 5 CDBP ffffff89d1b44a08, reason 0x0 (COMPLETED) pkt_state
0x1f state 0x0 Success Time 6859us
00000.103251793 mpt0:<- 0x28  READ(10) address 09:00, lba 0x0edd4e4e, len 0x0001 00, c
ontrol 0x00 timeout 5 CDBP ffffffc5e9f5e9a8, reason 0x0 (COMPLETED) pkt_sta te 0x1f st
ate 0x0 Success Time 6840us
```

The SCSI read I/O is first issued by the scsi_vhci driver: the SCSI virtual host controller interconnect driver. This driver presents multiple paths to disks as single virtual targets. The SCSI I/O is then processed by the mpt driver: the SCSI host bus adapter driver. The mpt driver sends the I/O to the correct path on the host bus adapter card, which then sends it to the external storage device.

Latency by Driver Instance. Apart from tracing SCSI commands `iosnoop`-style, the `scsi.d` script can also summarize data and print reports. The following was run on a system performing a streaming I/O workload to an external storage JBOD, connected using two paths and configured with multipathing:

```
solaris# scsi.d -D QUIET -D PERF_REPORT
Hit Control C to interrupt
^C

  nv_sata                                                    4
           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |@@@@@@@                                  57
          262144 |@@@@@@@@@@@@                             97
          524288 |@@@@@@@@@@@                              93
         1048576 |@@@@@@@                                  59
         2097152 |                                         2
         4194304 |                                         4
         8388608 |                                         2
        16777216 |                                         3
        33554432 |                                         3
        67108864 |                                         4
       134217728 |                                         0

  mpt                                                        0
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |                                         125
         1048576 |                                         578
         2097152 |@@                                       4900
```

```
         4194304 |@@@@                                     14493
         8388608 |@@@@@@@@@                                29044
        16777216 |@@@@@@@@@@@@@@@@@                        56658
        33554432 |@@@@@@                                   19138
        67108864 |@                                        4258
       134217728 |                                         515
       268435456 |                                         20
       536870912 |                                         0

  mpt                                                       2
           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |                                         1
          524288 |                                         125
         1048576 |                                         551
         2097152 |@                                        4829
         4194304 |@@@@                                     14448
         8388608 |@@@@@@@@@                                29081
        16777216 |@@@@@@@@@@@@@@@@@                        56614
        33554432 |@@@@@@                                   19186
        67108864 |@                                        4343
       134217728 |                                         527
       268435456 |                                         14
       536870912 |                                         0

  scsi_vhci                                                 0
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |                                         250
         1048576 |                                         1125
         2097152 |@                                        9717
         4194304 |@@@@                                     28945
         8388608 |@@@@@@@@@                                58110
        16777216 |@@@@@@@@@@@@@@@@@                        113295
        33554432 |@@@@@@                                   38329
        67108864 |@                                        8601
       134217728 |                                         1042
       268435456 |                                         34
       536870912 |                                         0
```

The output shows driver and instance number, with a distribution plot of SCSI command response times in nanoseconds. The first plot for nv_sata instance 4 (often written as nv_sata4) is for the internal system disks, which are not performing much I/O (although some outliers reached the 67-ms to 134-ms range).

The plots for mpt0 and mpt2 show the SCSI I/O on the mpt driver, which is the SCSI host bus adapter driver. There are two instances, one for each path, allowing two useful observations to be made.

The I/O count between the two paths appears well balanced (total counts appear similar). If there was a problem with the multipathing driver (mpxio) favoring one path over the other (which should not occur), the counts would differ.

The latency between the two paths also appears similar, with the mpt0 path performing 56,658 SCSI commands 16 ms to 33 ms range and mpt2 performing 56,614 in the same range. If one path was slower than the other, this could be evidence of a hardware issue with the cabling and remote storage controllers.

The last plot is for scsi_vhci, the driver exporting virtual sd instances that represent multipathed drives. The counts for scsi_vhci appear to sum both mpt paths, which is expected.

SATA Scripts

These use the fbt provider to trace the SATA and SAS drivers. DTracing these drivers can provide more lower-level details of disk I/O operation than with the io provider alone. Functionally, the SATA IO stack works as shown in Figure 4-6.

The high-level diagram is in the "Capabilities" section.

Since there is currently no stable SATA provider, the fbt[15] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for SATA analysis. Table 4-7 is a Solaris SATA driver reference.

See the man pages for each driver for the complete description.

SATA drivers are a complex and low-level part of the kernel; DTracing them using the fbt provider exposes this complexity. However, you don't need to be a kernel engineer on Oracle's SATA driver team to understand or write these scripts:

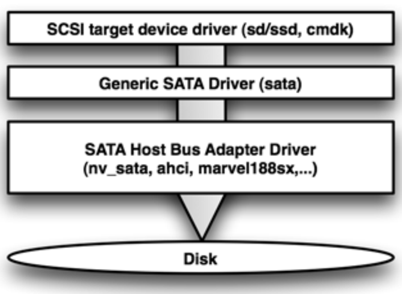


Figure 4-6 SATA I/O stack

15. See the "fbt Provider" section in Chapter 12 for more discussion about use of the fbt provider.

Table 4-7 Solaris SATA Driver Reference

Driver	Synopsis	Description
sata	Solaris SATA framework	Serial ATA is an interconnect technology designed to replace parallel ATA technology. It is used to connect hard drives, optical drives, removable magnetic media devices, and other peripherals to the host system. For complete information on Serial ATA technology, visit the Serial ATA Web site at *www.serialata.org*.
nv_sata	Nvidia ck804/ mcp55 SATA controller driver	The nv_sata driver is a SATA HBA driver that supports Nvidia ck804 and mcp55 SATA HBA controllers. While these Nvidia controllers support standard SATA features including SATA-II drives, NCQ, hotplug, and ATAPI drives, the driver currently does not support NCQ features.
ahci	Advanced Host Controller Interface SATA controller driver	The ahci driver is a SATA framework-compliant HBA driver that supports SATA HBA controllers compatible with the Advanced Host Controller Interface 1.0 specification. AHCI is an Intel-developed protocol that describes the register-level interface for host controllers for Serial ATA 1.0a and Serial ATA II. The AHCI 1.0 specification describes the interface between the system software and the host controller hardware.
marvell88sx	Marvell 88SX SATA controller driver	The marvell88sx driver is a SATA framework-compliant HBA driver supporting the Marvell 88SX5081, 88SX5080, 88SX5040, 88SX5041, 88SX6081, and 88SX6041 controllers.

There are tricks and techniques for using DTrace and experimentation to gain quick familiarity with an unknown subsystem. The case study at the end of this chapter demonstrates this: DTracing an unfamiliar I/O driver (SATA).

satacmds.d

A good place to start with any target is to DTrace high-level information. In this example, we count the SATA commands being issued.

Script

Most of this script is the `dtrace:::BEGIN` statement, which defines an associative array for translating SATA command codes into human-readable strings. This block of code was autogenerated by processing a SATA definitions header file.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

string sata_cmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * These are from the SATA_DIR_* and SATA_OPMODE_* definitions in
         * /usr/include/sys/sata/sata_hba.h:
         */
        sata_dir[1] = "no-data";
        sata_dir[2] = "read";
        sata_dir[4] = "write";
        sata_opmode[0] = "ints+async";  /* interrupts and asynchronous */
        sata_opmode[1] = "poll";        /* polling */
        sata_opmode[4] = "synch";       /* synchronous */
        sata_opmode[5] = "synch+poll";  /* (valid?) */

        /*
         * These SATA command descriptions were generated from the SATAC_*
         * definitions in /usr/include/sys/sata/sata_defs.h:
         */
        sata_cmd[0x90] = "diagnose command";
        sata_cmd[0x10] = "restore cmd, 4 bits step rate";
        sata_cmd[0x50] = "format track command";
        sata_cmd[0xef] = "set features";
        sata_cmd[0xe1] = "idle immediate";
        sata_cmd[0xe0] = "standby immediate";
        sata_cmd[0xde] = "door lock";
        sata_cmd[0xdf] = "door unlock";
        sata_cmd[0xe3] = "idle";
        sata_cmd[0xe2] = "standby";
        sata_cmd[0x08] = "ATAPI device reset";
        sata_cmd[0x92] = "Download microcode";
        sata_cmd[0xed] = "media eject";
        sata_cmd[0xe7] = "flush write-cache";
        sata_cmd[0xec] = "IDENTIFY DEVICE";
        sata_cmd[0xa1] = "ATAPI identify packet device";
        sata_cmd[0x91] = "initialize device parameters";
        sata_cmd[0xa0] = "ATAPI packet";
        sata_cmd[0xc4] = "read multiple w/DMA";
        sata_cmd[0x20] = "read sector";
        sata_cmd[0x40] = "read verify";
        sata_cmd[0xc8] = "read DMA";
        sata_cmd[0x70] = "seek";
        sata_cmd[0xa2] = "queued/overlap service";
        sata_cmd[0xc6] = "set multiple mode";
        sata_cmd[0xca] = "write (multiple) w/DMA";
        sata_cmd[0xc5] = "write multiple";
        sata_cmd[0x30] = "write sector";
        sata_cmd[0x24] = "read sector extended (LBA48)";
        sata_cmd[0x25] = "read DMA extended (LBA48)";
        sata_cmd[0x29] = "read multiple extended (LBA48)";
        sata_cmd[0x34] = "write sector extended (LBA48)";
        sata_cmd[0x35] = "write DMA extended (LBA48)";
        sata_cmd[0x39] = "write multiple extended (LBA48)";
        sata_cmd[0xc7] = "read DMA / may be queued";
        sata_cmd[0x26] = "read DMA ext / may be queued";
        sata_cmd[0xcc] = "write DMA / may be queued";
        sata_cmd[0x36] = "write DMA ext / may be queued";
        sata_cmd[0xe4] = "read port mult reg";
        sata_cmd[0xe8] = "write port mult reg";
        sata_cmd[0x60] = "First-Party-DMA read queued";
```

```
66          sata_cmd[0x61] = "First-Party-DMA write queued";
67          sata_cmd[0x2f] = "read log";
68          sata_cmd[0xb0] = "SMART";
69          sata_cmd[0xe5] = "check power mode";
70
71          printf("Tracing... Hit Ctrl-C to end.\n");
72  }
73
74  /*
75   * Trace SATA command start by probing the entry to the SATA HBA driver.  Four
76   * different drivers are covered here; add yours here if it is missing.
77   */
78  fbt::nv_sata_start:entry,
79  fbt::bcm_sata_start:entry,
80  fbt::ahci_tran_start:entry,
81  fbt::mv_start:entry
82  {
83          this->dev = (struct dev_info *)arg0;
84          this->sata_pkt = (sata_pkt_t *)arg1;
85
86          this->modname = this->dev != NULL ?
87              stringof(this->dev->devi_node_name) : "<unknown>";
88          this->dir = this->sata_pkt->satapkt_cmd.satacmd_flags.sata_data_direction;
89          this->dir_text = sata_dir[this->dir] != NULL ?
90              sata_dir[this->dir] : "<none>";
91          this->cmd = this->sata_pkt->satapkt_cmd.satacmd_cmd_reg;
92          this->cmd_text = sata_cmd[this->cmd] != NULL ?
93              sata_cmd[this->cmd] : lltostr(this->cmd);
94          this->op_mode = this->sata_pkt->satapkt_op_mode;
95          this->op_text = sata_opmode[this->op_mode] != NULL ?
96              sata_opmode[this->op_mode] : lltostr(this->op_mode);
97
98          @[this->modname, this->dir_text, this->cmd_text, this->op_text] =
99              count();
100 }
101
102 dtrace:::END
103 {
104         printf("  %-14s %-9s %-30s %-10s   %s\n", "DEVICE NODE", "DIR",
105             "COMMAND", "OPMODE", "COUNT");
106         printa("  %-14s %-9s %-30s %-10s   %@d\n", @);
107 }
```

Script satacmds.d

This script traces SATA commands by tracing the individual SATA HBA driver (the `sata_tran_start` function). This function has two arguments: a `struct dev_info` and `sata_pkt_t`. To figure out how to fetch various information from `sata_pkt_t`, structure definitions were examined in the kernel source code and then examined using DTrace along with known workloads.

If this script fails to trace any SATA commands on your system, find the appropriate probe for the start function in your SATA HBA driver, and add it after line 81. If that fails, you can use the higher-level function `sata_hba_start()` by replacing lines 78 through to 84 with this:

```
fbt::sata_hba_start:entry
{
        this->hba_inst = args[0]->txlt_sata_hba_inst;
        this->sata_pkt = args[0]->txlt_sata_pkt;

        this->dev = (struct dev_info *)this->hba_inst->satahba_dip;

```

`sata_hba_start()` is from the generic sata driver, and as such makes for a script that is more likely to see SATA commands on different systems. The downside is that `sata_hba_start()` traces most, but not all, SATA commands.

Warning

The `sata_cmd` translations in this script were automatically generated from the definitions in the `/usr/include/sys/sata/sata_defs.h` file, for example:

```
#define SATAC_READ_DMA_QUEUED   0xc7    /* read DMA / may be queued */
#define SATAC_READ_DMA_QUEUED_EXT 0x26  /* read DMA ext / may be queued */
#define SATAC_WRITE_DMA_QUEUED  0xcc    /* read DMA / may be queued */
#define SATAC_WRITE_DMA_QUEUED_EXT 0x36 /* read DMA ext / may be queued */
```

The `sed` utility was used to strip out all text apart from the hexadecimal value and the comment and replace them with the D syntax for an associative array declaration. For sed programmers who are curious, I copied the `#defines` to a `cmds.h` file and then used the following:

```
# sed 's:[^/]*0x:     sata_cmd[0x:;s:[        ]*/\* :] = ":;s/...$/";/' cmds.h
[...]
        sata_cmd[0xc7] = "read DMA / may be queued";
        sata_cmd[0x26] = "read DMA ext / may be queued";
        sata_cmd[0xcc] = "read DMA / may be queued";
        sata_cmd[0x36] = "read DMA ext / may be queued";
[...]
```

I then copied this output into the `scsicmds.d` script. This is an example of using one programming language (sed) to generate another programming language (D).

But not so fast: Take a closer look at the four lines in the previous samples. Should the comment for `SATAC_WRITE_DMA_QUEUED` really be "read DMA / may be queued"? Shouldn't this be "write DMA / may be queued"? This looks like a copy-and-paste error.

That raises an important warning: There are not only bugs in kernel code (as in all source code), but there are also bugs in source code *comments*. Be a little cautious when reading them to understand some code or when processing them, as shown earlier.

Examples

Examples include read I/O, synchronous vs. asyncronous write workloads, UFS vs. ZFS synchronous writes, and device insertion.

Read I/O. A read I/O workload was performed to local system disks, attached via SATA (the nv_sata driver). The following shows the SATA commands issued:

```
solaris# satacmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE    DIR        COMMAND                        OPMODE       COUNT
  pci10de,cb84   no-data    flush write-cache              ints+async   2
  pci10de,cb84   write      write DMA extended (LBA48)     ints+async   2
  pci10de,cb84   read       read DMA extended (LBA48)      ints+async   1988
```

The most frequent SATA command issued was reads via DMA with the default opmode (interrupts and asynchronous).

The device node pci10de,cb84 is the path to the SATA HBA for these local system disks. This is perhaps the easiest indicator of the device that can be extracted from the available probe arguments; more information about the device can be obtained by enhancing the script (though it will quickly become complex).

Synchronous vs. Asyncronous Write Workloads. Synchronous writes are where the system waits until the write has been written to stable storage before returning a completion. Applications can request synchronous semantics by opening files with a `SYNC` flag (often `O_DSYNC`). They are sometimes used by applications to ensure that critical data is known to be written before moving on, such as when databases write their log files.

Here a synchronous write workload was performed on a ZFS file system using local system disks:

```
solaris# satacmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE    DIR        COMMAND                        OPMODE       COUNT
  pci10de,cb84   no-data    flush write-cache              ints+async   752
  pci10de,cb84   write      write DMA extended (LBA48)     ints+async   752
```

The output shows how ZFS is implementing synchronous writes. Writes are issued, along with an equal number of `flush write-cache` requests. The `flush write-cache` request ensures that the disk has written its onboard write cache to stable storage, which ZFS is calling (via a `DKIOCFLUSHWRITECACHE` ioctl) to ensure that the disk really has written out the data. Without this `flush write-cache`, the regular write command would return quickly from disk as it was cached on disk-based DRAM, and the application would think that the write completed, but it hasn't until the disk itself flushes its own cache. Should a power outage occur before the disk can do this, data is lost even though the file system and application believe it to have been written. This is data corruption. ZFS ensures that this does not happen by waiting for flush write-cache requests before believing that the disk has really written its data.

The write DMA extended command specifies how the write should be performed: Write the data via Direct Memory Access.

Compare the synchronous write workload with an asynchoronous write workload, which is the default when writing data to file systems:

```
solaris# satacmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE    DIR       COMMAND                        OPMODE       COUNT
  pci10de,cb84   read      read DMA extended (LBA48)      ints+async   3
  pci10de,cb84   no-data   flush write-cache              ints+async   18
  pci10de,cb84   write     write DMA extended (LBA48)     ints+async   2395
```

UFS vs. ZFS Synchronous Writes. In the previous example, SATA commands called by the ZFS file system performing synchronous writes were shown. Let's try the same with the UFS file system:

```
solaris# satacmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE    DIR       COMMAND                        OPMODE       COUNT
  pci10de,cb84   write     write DMA extended (LBA48)     ints+async   1918
```

What's this? No flush write-cache commands? As previously mentioned, this is a problem (and is true on many other file systems), because disks can and will buffer write data and send completion interrupts before they have actually written to stable storage. If a power outage occurs at the wrong time, data corruption can occur because the file system may think that it has written data that it has not. Remember how UFS required the fsck(1M) tool to repair file system corruption and inconsistencies? ZFS does not—it cannot become corrupted or have its on-disk state made inconsistent. This is part of the reason why.

Device Insertion. Here a SATA device was inserted while satacmds.d was tracing:

```
solaris# satacmds.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE    DIR       COMMAND                         OPMODE       COUNT
  pci10de,cb84   no-data   set features                    synch        1
  pci10de,cb84   no-data   idle                            synch        2
  pci10de,cb84   no-data   check power mode                synch        3
  pci10de,cb84   no-data   read verify                     synch        3
  pci10de,cb84   read      IDENTIFY DEVICE                 synch        3
  pci10de,cb84   read      read DMA extended (LBA48)       ints+async   3
  pci10de,cb84   no-data   flush write-cache               ints+async   38
  pci10de,cb84   write     write DMA extended (LBA48)      ints+async   78
```

Various different SATA commands can be seen that were sent to initialize the new device, including `IDENTIFY DEVICE` and `set features`. Note that the operation mode for these commands is synchronous.

satarw.d

This script takes the three most common SATA commands (`read`, `write`, and `sync-cache`) and prints summaries of the I/O sizes and times. It's simple but effective—this provides visibility for the I/O throughput rates and latencies for SATA.

Script

The start probes for the SATA events were traced in the generic sata driver using `sata_txlt_read()`, and so on. Latency time is calculated by storing the start time for these SATA commands in an associative array keyed on the sata packet address and retrieved when the SATA command is completed. The trickiest part was fetching the size of the I/O, which was much easier to pull from the SCSI layer than from SATA.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /*
         * SATA_DIR of type 1 normally means no-data, but we can call it
         * sync-cache as that's the only type 1 we are tracing.
         */
         sata_dir[1] = "sync-cache";
         sata_dir[2] = "read";
         sata_dir[4] = "write";

         printf("Tracing... Hit Ctrl-C to end.\n");
 }

 /* cache the I/O size while it is still easy to determine */
```

continues

```
fbt::sd_start_cmds:entry
{
        /* see the sd_start_cmds() source to understand the following logic */
        this->bp = args[1] != NULL ? args[1] : args[0]->un_waitq_headp;
        self->size = this->bp != NULL ? this->bp->b_bcount : 0;
}

fbt::sd_start_cmds:return { self->size = 0; }

/* trace generic SATA driver functions for read, write and sync-cache */
fbt::sata_txlt_read:entry,
fbt::sata_txlt_write:entry,
fbt::sata_txlt_synchronize_cache:entry
{
        this->sata_pkt = args[0]->txlt_sata_pkt;
        start[(uint64_t)this->sata_pkt] = timestamp;
        size[(uint64_t)this->sata_pkt] = self->size;
}

/* SATA command completed */
fbt::sata_pkt_free:entry
/start[(uint64_t)args[0]->txlt_sata_pkt]/
{
        this->sata_pkt = args[0]->txlt_sata_pkt;
        this->delta = (timestamp - start[(uint64_t)this->sata_pkt]) / 1000;
        this->size = size[(uint64_t)this->sata_pkt];
        this->dir = this->sata_pkt->satapkt_cmd.satacmd_flags.sata_data_direction;
        this->dir_text = sata_dir[this->dir] != NULL ?
            sata_dir[this->dir] : "<none>";

        @num[this->dir_text] = count();
        @avg_size[this->dir_text] = avg(this->size);
        @avg_time[this->dir_text] = avg(this->delta);
        @sum_size[this->dir_text] = sum(this->size);
        @sum_time[this->dir_text] = sum(this->delta);
        @plot_size[this->dir_text] = quantize(this->size);
        @plot_time[this->dir_text] = quantize(this->delta);

        start[(uint64_t)this->sata_pkt] = 0;
        size[(uint64_t)this->sata_pkt] = 0;
}

dtrace:::END
{
        normalize(@avg_size, 1024);
        normalize(@sum_size, 1048576);
        normalize(@sum_time, 1000);
        printf("  %-10s  %10s  %10s %10s  %10s %12s\n", "DIR",
            "COUNT", "AVG(KB)", "TOTAL(MB)", "AVG(us)", "TOTAL(ms)");
        printa("  %-10s  %@10d  %@10d %@10d  %@10d %@12d\n", @num,
            @avg_size, @sum_size, @avg_time, @sum_time);
        printf("\n\nSATA I/O size (bytes):\n");
        printa(@plot_size);
        printf("\nSATA I/O latency (us):\n");
        printa(@plot_time);
}
```

Script satarw.d

Example

This shows a mixed workload of reads and synchronous writes, on a ZFS file system:

```
solaris# satarw.d
Tracing... Hit Ctrl-C to end.
^C
  DIR                 COUNT      AVG(KB)   TOTAL(MB)      AVG(us)     TOTAL(ms)
  sync-cache            914            0           0         9187          8397
  write                 914           12          10          198           181
  read                 1091            6           6         1773          1935


SATA I/O size (bytes):

  sync-cache
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 914
               1 |                                         0

  read
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@                      520
            1024 |                                         8
            2048 |@@@@@@@@                                 221
            4096 |@@@@@@                                   171
            8192 |@@@                                      95
           16384 |@@                                       41
           32768 |                                         8
           65536 |                                         9
          131072 |@                                        18
          262144 |                                         0

  write
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 914
           16384 |                                         0


SATA I/O latency (us):

  write
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 903
             256 |                                         8
             512 |                                         3
            1024 |                                         0

  read
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |@@@@@@@@@@                               275
             128 |@@@@@@@@@@@@@@@@@@@@@                    576
             256 |@                                        30
             512 |                                         9
            1024 |                                         10
            2048 |@                                        20
            4096 |@@                                       63
            8192 |@@@@                                     96
```

continues

```
           16384 |                                         11
           32768 |                                         1
           65536 |                                         0

  sync-cache
           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@                                        21
            2048 |@@                                       50
            4096 |@@@@@@@@@@                               233
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@                578
           16384 |@                                        29
           32768 |                                         3
           65536 |                                         0
```

The `sync-cache` commands show zero bytes, since they are not performing data transfer.

satareasons.d

When SATA commands complete, a reason code is set to show why the completion was sent. Typically most SATA commands will succeed, and the reason code returned will be "Success." If SATA commands are erroring, the reason code can be useful to examine to understand the type of error.

Script

`satareasons.d` includes translations of SATA reasons codes. To keep the script length down, only a handful of common SATA command codes are translated:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

string sata_cmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * These are SATA_DIR_* from /usr/include/sys/sata/sata_hba.h:
         */
        sata_dir[1] = "no-data";
        sata_dir[2] = "read";
        sata_dir[4] = "write";

        /*
         * Some SATAC_* definitions from /usr/include/sys/sata/sata_defs.h, for
         * commands commonly issued.  More can be added from satacmds.d.
         */
        sata_cmd[0x20] = "read sector";
        sata_cmd[0x25] = "read DMA extended";
        sata_cmd[0x35] = "write DMA extended";
        sata_cmd[0x30] = "write sector";
        sata_cmd[0x40] = "read verify";
        sata_cmd[0x70] = "seek";
        sata_cmd[0x90] = "diagnose command";
        sata_cmd[0xb0] = "SMART";
```

```
        sata_cmd[0xec] = "IDENTIFY DEVICE";
        sata_cmd[0xe5] = "check power mode";
        sata_cmd[0xe7] = "flush write-cache";
        sata_cmd[0xef] = "set features";

        /*
         * These are SATA_PKT_* from /usr/include/sys/sata/sata_hba.h:
         */
        sata_reason[-1] = "Not completed, busy";
        sata_reason[0] = "Success";
        sata_reason[1] = "Device reported error";
        sata_reason[2] = "Not accepted, queue full";
        sata_reason[3] = "Not completed, port error";
        sata_reason[4] = "Cmd unsupported";
        sata_reason[5] = "Aborted by request";
        sata_reason[6] = "Operation timeout";
        sata_reason[7] = "Aborted by reset request";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::sd_start_cmds:entry
{
        /* see the sd_start_cmds() source to understand the following logic */
        self->bp = args[1] != NULL ? args[1] : args[0]->un_waitq_headp;
}

fbt::sd_start_cmds:return { self->bp = 0; }

fbt::sata_hba_start:entry
/self->bp->b_dip/
{
        statname[args[0]->txlt_sata_pkt] =
            xlate <devinfo_t *>(self->bp)->dev_statname;
}

fbt::sata_pkt_free:entry
/args[0]->txlt_sata_pkt->satapkt_cmd.satacmd_cmd_reg/
{
        this->sata_pkt = args[0]->txlt_sata_pkt;
        this->devname = statname[this->sata_pkt] != NULL ?
            statname[this->sata_pkt] : "<?>";
        this->dir = this->sata_pkt->satapkt_cmd.satacmd_flags.sata_data_direction;
        this->dir_text = sata_dir[this->dir] != NULL ?
            sata_dir[this->dir] : "<none>";
        this->cmd = this->sata_pkt->satapkt_cmd.satacmd_cmd_reg;
        this->cmd_text = sata_cmd[this->cmd] != NULL ?
            sata_cmd[this->cmd] : lltostr(this->cmd);
        this->reason = this->sata_pkt->satapkt_reason;
        this->reason_text = sata_reason[this->reason] != NULL ?
            sata_reason[this->reason] : lltostr(this->reason);
        statname[this->sata_pkt] = 0;

        @[this->devname, this->dir_text, this->cmd_text, this->reason_text] =
            count();
}

dtrace:::END
{
        printf("  %-8s %-10s %-20s %25s  %s\n", "DEVICE", "DIR", "COMMAND",
            "REASON", "COUNT");
        printa("  %-8s %-10s %-20s %25s  %@d\n", @);
}
```

Script satareasons.d

The script gets a little complex because it has to fetch the name of the disk device and does so by recording an associative array called `statname` that translates SATA packet addresses into device names. The SATA command completions are traced by the `sata_pkt_free()` function, because it is the end of the road for that SATA command—at this point the packet is freed. Details are not tracked in `sata_pkt_free()` if the SATA command was zero, which may be because of an invalid packet that was never sent anyway (and is now being freed).

Examples

The SATA disk sd4 was during read I/O:

```
solaris# satareasons.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE   DIR        COMMAND                                    REASON  COUNT
  <?>      read       read DMA extended                         Success  1
  sd4      read       read DMA extended    Not completed, port error  1
  sd6      write      write DMA extended                        Success  46
  sd7      write      write DMA extended                        Success  46
  <?>      no-data    flush write-cache                         Success  58
  sd4      read       read DMA extended                         Success  6704
```

One of the read commands on sd4 returned `Not completed, port error`, because the disk had been removed.

satalatency.d

The `satalatency.d` script summarizes SATA command latency in terms of SATA command and completion reasons.

Script

Time is measured from the entry to the SATA HBA driver to when the generic SATA driver frees the SATA packet.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

string sata_cmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * Some SATAC_* definitions from /usr/include/sys/sata/sata_defs.h, for
         * commands commonly issued.  More can be added from satacmds.d.
         */
        sata_cmd[0x20] = "read sector";
        sata_cmd[0x25] = "read DMA extended";
        sata_cmd[0x35] = "write DMA extended";
```

```
        sata_cmd[0x30] = "write sector";
        sata_cmd[0x40] = "read verify";
        sata_cmd[0x70] = "seek";
        sata_cmd[0x90] = "diagnose command";
        sata_cmd[0xb0] = "SMART";
        sata_cmd[0xec] = "IDENTIFY DEVICE";
        sata_cmd[0xe5] = "check power mode";
        sata_cmd[0xe7] = "flush write-cache";
        sata_cmd[0xef] = "set features";

        /*
         * These are SATA_PKT_* from /usr/include/sys/sata/sata_hba.h:
         */
        sata_reason[-1] = "Not completed, busy";
        sata_reason[0] = "Success";
        sata_reason[1] = "Device reported error";
        sata_reason[2] = "Not accepted, queue full";
        sata_reason[3] = "Not completed, port error";
        sata_reason[4] = "Cmd unsupported";
        sata_reason[5] = "Aborted by request";
        sata_reason[6] = "Operation timeout";
        sata_reason[7] = "Aborted by reset request";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

/*
 * Trace SATA command start by probing the entry to the SATA HBA driver.  Four
 * different drivers are covered here; add yours here if it is missing.
 */
fbt::nv_sata_start:entry,
fbt::bcm_sata_start:entry,
fbt::ahci_tran_start:entry,
fbt::mv_start:entry
{
        start[arg1] = timestamp;
}

fbt::sata_pkt_free:entry
/start[(uint64_t)args[0]->txlt_sata_pkt]/
{
        this->sata_pkt = args[0]->txlt_sata_pkt;
        this->delta = (timestamp - start[(uint64_t)this->sata_pkt]) / 1000;
        this->cmd = this->sata_pkt->satapkt_cmd.satacmd_cmd_reg;
        this->cmd_text = sata_cmd[this->cmd] != NULL ?
            sata_cmd[this->cmd] : lltostr(this->cmd);
        this->reason = this->sata_pkt->satapkt_reason;
        this->reason_text = sata_reason[this->reason] != NULL ?
            sata_reason[this->reason] : lltostr(this->reason);

        @num[this->cmd_text, this->reason_text] = count();
        @average[this->cmd_text, this->reason_text] = avg(this->delta);
        @total[this->cmd_text, this->reason_text] = sum(this->delta);

        start[(uint64_t)this->sata_pkt] = 0;
}

dtrace:::END
{
        normalize(@total, 1000);
        printf("\n  %-18s %23s %10s %10s %10s\n", "SATA COMMAND",
            "COMPLETION", "COUNT", "AVG(us)", "TOTAL(ms)");
        printa("  %-18s %23s %@10d %@10d %@10d\n", @num, @average, @total);
}
```

Script chpt_diskIO_satalatency.d

Example

The following system was performing a mixed workload of random disk reads and synchronous writes on a ZFS file system:

```
solaris# satalatency.d
Tracing... Hit Ctrl-C to end.
^C

  SATA COMMAND                    COMPLETION        COUNT      AVG(us)    TOTAL(ms)
  flush write-cache                  Success          728         9131         6647
  write DMA extended                 Success          729          222          162
  read DMA extended                  Success         3488         4187        14607
```

Based on just the average time for the read and write commands, it would appear that the read I/O are the slowest, with an average time of 4.1 ms, whereas writes returned in 0.2 ms. However, since this is synchronous writes on ZFS, the write will not be acknowledged to the application until the `flush write-cache` command has completed, which averages 9.1 ms.

IDE Scripts

These use the fbt provider to trace the IDE driver. DTracing the IDE driver can provide lower-level details of disk I/O operation than with the io provider alone. Functionally, the IDE IO stack works as shown in Figure 4-7.

Also see the high-level diagram in the "Capabilities" section.

Since there is currently no stable IDE provider, the fbt[16] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may

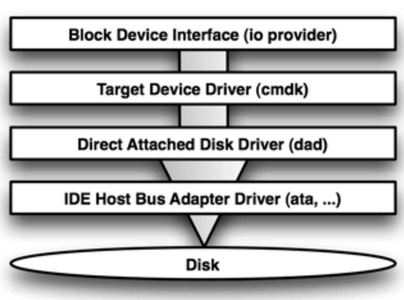


Figure 4-7 IDE I/O stack

16. See the "fbt Provider" section in Chapter 12 for more discussion about use of the fbt provider.

Table 4-8 Solaris IDE Driver Reference

Driver	Synopsis	Description
cmdk	Common disk driver	A common interface to various disk devices. The driver supports magnetic fixed disks and magnetic removable disks.
dad	Driver for IDE disk devices	Handles the IDE disk drives on SPARC platforms. The type of disk drive is determined using the ATA IDE identify device command and by reading the volume label stored on the drive. The dad device driver supports the Solaris SPARC VTOC and the EFI/GPT disk volume labels.
ata	AT attachment disk driver	Supports disk and ATAPI CD/DVD devices conforming to the AT Attachment specification including IDE interfaces. Support is provided for both parallel ATA (PATA) and serial ATA (SATA) interfaces.

change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for IDE analysis.

See the man pages for each driver for the complete description.

Familiarization

To become familiar with DTracing IDE, we'll perform 10,000 read disk I/O to an IDE disk:

```
# dd if=/dev/rdsk/c0d0s0 of=/dev/null count=10000
```

We use DTrace to frequency count all calls to the dad driver:

```
solaris# dtrace -n 'fbt:dadk::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:dadk::entry ' matched 43 probes
^C

  dadk_getgeom                                              1
  dadk_getphygeom                                           2
  dadk_iob_alloc                                            5
  dadk_iob_free                                             5
  dadk_iob_xfer                                             5
  dadk_strategy                                         10000
  dadk_iodone                                           10005
  dadk_ioprep                                           10005
  dadk_iosetup                                          10005
```

continues

```
dadk_pkt                                                    10005
dadk_pktcb                                                  10005
dadk_pktprep                                                10005
dadk_transport                                              10005
```

The `dadk_strategy()` and `dadk_iodone()` are familiar (`bdev_strategy()` and `biodone()`), but they only take `buf_t` as the argument; we can already trace such `buf_t` details using the stable io provider, as shown earlier in this chapter. If we are digging into the IDE driver, it's to see specific IDE command information, such as is available in the following functions:

```
dadk_iosetup(struct dadk *dadkp, struct cmpkt *pktp)
dadk_pktcb(struct cmpkt *pktp)
```

The first can be used to trace regular IDE I/O, and the second can be used for IDE command completions (`pktcb` == packet callback). Both have access to specific IDE information in their arguments. `dadk_iosetup()` does miss `ioctl()`, which needs to be traced separately (`dadk_ioctl()` or something deeper along the code path, after IDE information has been initialized).

Because IDE is less commonly used these days, only a few IDE scripts are included here, chosen for the widest coverage. They can be customized as needed into scripts similar to those in the "SATA" section.

idelatency.d

Because of dad's simple interface, this script is one of the most straightforward in this chapter's collection of kernel driver scripts:

Script

To record the time between IDE command start and completion events, a packet pointer was used in an associative array to store the time stamp by packet. This was easy to retrieve in `dadk_iosetup` (`arg1`) but not so easy in `dadk_ioctl()` because the packet hasn't been created yet; to solve this, `dadk_pktprep()` is traced as the starting point for both types of IDE command, because it is common to both code paths.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

string dcmd[uchar_t];

dtrace:::BEGIN
{
```

```
        /*
         * These command descriptions are from the DCMD_* definitions
         * in /usr/include/sys/dktp/dadkio.h:
         */
        dcmd[1] = "Read Sectors/Blocks";
        dcmd[2] = "Write Sectors/Blocks";
        dcmd[3] = "Format Tracks";
        dcmd[4] = "Format entire drive";
        dcmd[5] = "Recalibrate";
        dcmd[6] = "Seek to Cylinder";
        dcmd[7] = "Read Verify sectors on disk";
        dcmd[8] = "Read manufacturers defect list";
        dcmd[9] = "Lock door";
        dcmd[10] = "Unlock door";
        dcmd[11] = "Start motor";
        dcmd[12] = "Stop motor";
        dcmd[13] = "Eject medium";
        dcmd[14] = "Update geometry";
        dcmd[15] = "Get removable disk status";
        dcmd[16] = "cdrom pause";
        dcmd[17] = "cdrom resume";
        dcmd[18] = "cdrom play by track and index";
        dcmd[19] = "cdrom play msf";
        dcmd[20] = "cdrom sub channel";
        dcmd[21] = "cdrom read mode 1";
        dcmd[22] = "cdrom read table of contents header";
        dcmd[23] = "cdrom read table of contents entry";
        dcmd[24] = "cdrom read offset";
        dcmd[25] = "cdrom mode 2";
        dcmd[26] = "cdrom volume control";
        dcmd[27] = "flush write cache to physical medium";

        /* from CPS_* definitions in /usr/include/sys/dktp/cmpkt.h */
        reason[0] = "success";
        reason[1] = "failure";
        reason[2] = "fail+err";
        reason[3] = "aborted";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

/* IDE command start */
fbt::dadk_pktprep:return
{
        start[arg1] = timestamp;
}

/* IDE command completion */
fbt::dadk_pktcb:entry
/start[arg0]/
{
        this->pktp = args[0];

        this->delta = (timestamp - start[arg0]) / 1000;
        this->cmd = *((uchar_t *)this->pktp->cp_cdbp);
        this->cmd_text = dcmd[this->cmd] != NULL ?
            dcmd[this->cmd] : lltostr(this->cmd);
        this->reason = this->pktp->cp_reason;
        this->reason_text = reason[this->reason] != NULL ?
            reason[this->reason] : lltostr(this->reason);

        @num[this->cmd_text, this->reason_text] = count();
        @average[this->cmd_text, this->reason_text] = avg(this->delta);
        @total[this->cmd_text, this->reason_text] = sum(this->delta);
```

continues

```

        start[arg0] = 0;
}

dtrace:::END
{
        normalize(@total, 1000);
        printf("\n  %-36s %8s %8s %10s %10s\n", "IDE COMMAND",
            "REASON", "COUNT", "AVG(us)", "TOTAL(ms)");
        printa("  %-36s %8s %@8d %@10d %@10d\n", @num, @average, @total);
}
```

Script idelatency.d

Examples

Examples include known workload and synchronous ZFS writes.

Known Workload. In this example, a known load generating 10,000 disk reads was used:

```
solaris# idelatency.d
Tracing... Hit Ctrl-C to end.
^C

  IDE COMMAND                                REASON    COUNT    AVG(us)  TOTAL(ms)
  Read Sectors/Blocks                       success    10009        210       2105
```

The script output shows 10,009 IDE read commands (nine extra from other system activity), which had an average latency of 0.2 ms.

Synchronous ZFS Writes. The following was executed on a Solaris system with ZFS, performing a synchronous write workload:

```
solaris# idelatency.d
Tracing... Hit Ctrl-C to end.
^C

  IDE COMMAND                                REASON    COUNT    AVG(us)  TOTAL(ms)
  Read Sectors/Blocks                       success        7      30325        212
  flush write cache to physical medium      success      147      16848       2476
  Write Sectors/Blocks                      success     4087       4021      16435
```

As previously observed with SCSI and SATA, the latency for synchronous writes in ZFS is in the `sync cache` command, or, as it is described in the IDE header files, “flush write cache to physical medium,” which had an average latency of 16.8 ms.

iderw.d

This script takes the three common IDE commands (`read`, `write`, and `flush write-cache`) and prints summaries of the I/O sizes and times. This provides visibility for the I/O throughput rates and detailed latencies for IDE, especially outliers that may not be observable in the averages in `idelatency.d`.

Script

This script filters on the command type so that only the `reads`, `writes`, and `sync-cache` commands are traced:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

string dcmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * These commands of interest are from the DCMD_* definitions in
         * /usr/include/sys/dktp/dadkio.h:
         */
        dcmd[1] = "Read Sectors/Blocks";
        dcmd[2] = "Write Sectors/Blocks";
        dcmd[27] = "flush write cache";

        /* from CPS_* definitions in /usr/include/sys/dktp/cmpkt.h */
        reason[0] = "success";
        reason[1] = "failure";
        reason[2] = "fail+err";
        reason[3] = "aborted";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::dadk_pktprep:entry
{
        self->size = args[2]->b_bcount;
}

/* IDE command start */
fbt::dadk_pktprep:return
{
        start[arg1] = timestamp;
        size[arg1] = self->size;
        self->size = 0;
}

/* IDE command completion */
fbt::dadk_pktcb:entry
/start[arg0]/
{
        this->pktp = args[0];
        this->cmd = *((uchar_t *)this->pktp->cp_cdbp);
}

/* Only match desired commands: read/write/flush-cache */
fbt::dadk_pktcb:entry
```

continues

```
/start[arg0] && dcmd[this->cmd] != NULL/
{
        this->delta = (timestamp - start[arg0]) / 1000;
        this->cmd_text = dcmd[this->cmd] != NULL ?
            dcmd[this->cmd] : lltostr(this->cmd);
        this->size = size[arg0];

        @num[this->cmd_text] = count();
        @avg_size[this->cmd_text] = avg(this->size);
        @avg_time[this->cmd_text] = avg(this->delta);
        @sum_size[this->cmd_text] = sum(this->size);
        @sum_time[this->cmd_text] = sum(this->delta);
        @plot_size[this->cmd_text] = quantize(this->size);
        @plot_time[this->cmd_text] = quantize(this->delta);

        start[arg0] = 0;
        size[arg0] = 0;
}

dtrace:::END
{
        normalize(@avg_size, 1024);
        normalize(@sum_size, 1048576);
        normalize(@sum_time, 1000);
        printf("  %-20s  %8s  %10s %10s  %10s %11s\n", "DIR",
            "COUNT", "AVG(KB)", "TOTAL(MB)", "AVG(us)", "TOTAL(ms)");
        printa("  %-20s  %@8d  %@10d %@10d  %@10d %@11d\n", @num,
            @avg_size, @sum_size, @avg_time, @sum_time);
        printf("\n\nIDE I/O size (bytes):\n");
        printa(@plot_size);
        printf("\nIDE I/O latency (us):\n");
        printa(@plot_time);
}
```

Script iderw.d

Example

In this example, 100 1MB reads were performed on an IDE disk:

```
# dd if=/dev/rdsk/c1d0s0 of=/dev/null bs=1024k count=100
```

Tracing with `iderw.d`:

```
solaris# iderw.d
Tracing... Hit Ctrl-C to end.
^C
  DIR                      COUNT     AVG(KB)  TOTAL(MB)     AVG(us)   TOTAL(ms)
  Read Sectors/Blocks        405         252        100        1156         468


IDE I/O size (bytes):

  Read Sectors/Blocks
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |                                         5
            1024 |                                         0
```

```
            2048 |                                         0
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 400
          524288 |                                         0


IDE I/O latency (us):

  Read Sectors/Blocks
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |                                         2
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      358
            1024 |                                         0
            2048 |@@@@                                     40
            4096 |                                         1
            8192 |                                         3
           16384 |                                         1
           32768 |                                         0
```

The output may be surprising. The `dd` command requested 100 reads from the raw device (`/dev/rdsk`). Raw device interfaces are supposed to do as they are told, unlike block interfaces, which may use layers of caching. However, this is not what happened here: Despite `dd` requesting 100 1MB reads, the IDE disk actually performed this using 400 256KB reads.

This is because the IDE driver is respecting the `DK_MAXRECSIZE` constant from `/usr/include/sys/dktp/tgdk.h`:

```
#define    DK_MAXRECSIZE   (256<<10)       /* maximum io record size       */
```

ideerr.d

Errors are usually worth examining, and IDE has its own collection of possible error types. This script will print the IDE command, command completion reason, and (if available) additional IDE error information.

Script

The following script can be used to track error events in the IDE code path:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
```

continues

```
string dcmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * These command and error descriptions are from the DCMD_* and DERR_*
         * definitions in /usr/include/sys/dktp/dadkio.h:
         */

        dcmd[1] = "Read Sectors/Blocks";
        dcmd[2] = "Write Sectors/Blocks";
        dcmd[3] = "Format Tracks";
        dcmd[4] = "Format entire drive";
        dcmd[5] = "Recalibrate";
        dcmd[6] = "Seek to Cylinder";
        dcmd[7] = "Read Verify sectors on disk";
        dcmd[8] = "Read manufacturers defect list";
        dcmd[9] = "Lock door";
        dcmd[10] = "Unlock door";
        dcmd[11] = "Start motor";
        dcmd[12] = "Stop motor";
        dcmd[13] = "Eject medium";
        dcmd[14] = "Update geometry";
        dcmd[15] = "Get removable disk status";
        dcmd[16] = "cdrom pause";
        dcmd[17] = "cdrom resume";
        dcmd[18] = "cdrom play by track and index";
        dcmd[19] = "cdrom play msf";
        dcmd[20] = "cdrom sub channel";
        dcmd[21] = "cdrom read mode 1";
        dcmd[22] = "cdrom read table of contents header";
        dcmd[23] = "cdrom read table of contents entry";
        dcmd[24] = "cdrom read offset";
        dcmd[25] = "cdrom mode 2";
        dcmd[26] = "cdrom volume control";
        dcmd[27] = "flush write cache to physical medium";

        derr[0] = "success";
        derr[1] = "address mark not found";
        derr[2] = "track 0 not found";
        derr[3] = "aborted command";
        derr[4] = "write fault";
        derr[5] = "ID not found";
        derr[6] = "drive busy";
        derr[7] = "uncorrectable data error";
        derr[8] = "bad block detected";
        derr[9] = "invalid cdb";
        derr[10] = "hard device error- no retry";
        derr[11] = "Illegal length indication";
        derr[12] = "End of media detected";
        derr[13] = "Media change requested";
        derr[14] = "Recovered from error";
        derr[15] = "Device not ready";
        derr[16] = "Medium error";
        derr[17] = "Hardware error";
        derr[18] = "Illegal request";
        derr[19] = "Unit attention";
        derr[20] = "Data protection";
        derr[21] = "Miscompare";
        derr[22] = "Interface CRC error";
        derr[23] = "Reserved";

        /* from CPS_* definitions in /usr/include/sys/dktp/cmpkt.h */
        reason[0] = "success";
```

```
        reason[1] = "failure";
        reason[2] = "fail+err";
        reason[3] = "aborted";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::dadk_pktcb:entry
{
        this->pktp = args[0];

        this->cmd = *(char *)this->pktp->cp_cdbp;
        this->cmd_text = dcmd[this->cmd] != NULL ?
            dcmd[this->cmd] : lltostr(this->cmd);
        this->reason = this->pktp->cp_reason;
        this->reason_text = reason[this->reason] != NULL ?
            reason[this->reason] : lltostr(this->reason);
        this->err = *(char *)this->pktp->cp_scbp;
        this->err_text = derr[this->err] != NULL ?
            derr[this->err] : lltostr(this->err);

        @[this->cmd_text, this->reason_text, this->err_text] = count();
}

dtrace:::END
{
        printf("%-36s %8s %27s %s\n", "IDE COMMAND", "REASON", "ERROR",
            "COUNT");
        printa("%-36s %8s %27s %@d\n", @);
}
```

Script ideerr.d

If the IDE command returned with `CPS_CHKERR` (shown as `fail+err` to fit in the column; the full description is "command fails with status"), then there is an additional error code that must also be checked. The table for translating these is included in this script, `derr`, and they are printed in the `ERROR` column.

Example

On a system performing disk I/O to IDE disks:

```
# ideerr.d
IDE COMMAND                           REASON                        ERROR COUNT
Write Sectors/Blocks                 success                      success 136
Read Sectors/Blocks                  success                      success 20528
```

There are no errors—always good to see!

SAS Scripts

The fbt provider can be used to trace the SAS HBA drivers. Serial Attached SCSI (SAS) is a transport protocol usually used with external storage devices. DTracing

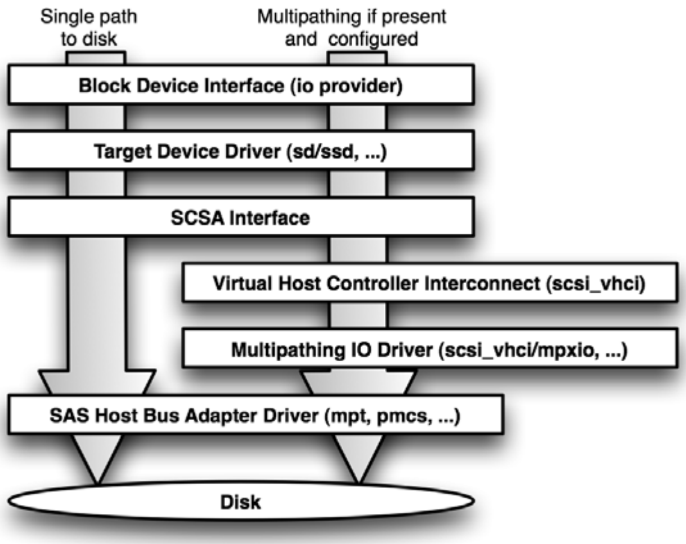


Figure 4-8 SAS I/O stack

SAS can provide lower-level details of disk I/O and SAS bus operation than the io provider alone can. Functionally, it can be described as in Figure 4-8.

Since there is currently no stable SAS provider, the fbt[17] and sdt providers are used. These are unstable interfaces: They expose kernel functions and data structures that may change from release to release. The scripts that follow were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for SAS analysis. Table 4-9 presents the Solaris SAS driver reference.

Table 4-9 Solaris SAS Driver Reference

Driver	Synopsis	Description
mpt	SCSI host bus adapter driver	A SCSA-compliant nexus driver that supports the LSI 53C1030 SCSI, SAS1064, SAS1068, and SAS1068E controllers.

17. See the "fbt Provider" section in Chapter 12 for more discussion about use of the fbt provider.

Familiarization

Unlike SCSI, there is no generic SAS driver that we can DTrace, so we must DTrace it in the specific HBA drivers that implement SAS. In the scripts that follow, the mpt driver is traced. If you would like to DTrace SAS on a different SAS HBA driver, the scripts will need to be rewritten to match its probes.

To get a quick insight into mpt internals, we performed an experiment where the dd(1) command issued 1,000 reads to a disk device, and all calls to mpt were frequency counted:

```
solaris# dtrace -n 'fbt:mpt::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:mpt::entry ' matched 287 probes
^C

  mpt_watch                                                           1
  mpt_capchk                                                          2
  mpt_scsi_setcap                                                     2
  mpt_ioc_faulted                                                     3
  mpt_watchsubr                                                       3
  mpt_sge_setup                                                    1005
  mpt_accept_pkt                                                   1007
  mpt_doneq_add                                                    1007
  mpt_intr                                                         1007
  mpt_prepare_pkt                                                  1007
  mpt_process_intr                                                 1007
  mpt_remove_cmd                                                   1007
  mpt_save_cmd                                                     1007
  mpt_scsi_destroy_pkt                                             1007
  mpt_scsi_init_pkt                                                1007
  mpt_scsi_start                                                   1007
  mpt_send_pending_event_ack                                       1007
  mpt_start_cmd                                                    1007
  mpt_check_acc_handle                                             2014
  mpt_doneq_rm                                                     2014
  mpt_free_extra_cmd_mem                                           2014
  mpt_doneq_empty                                                  4079
  mpt_check_dma_handle                                             6042
```

This shows some possibilities. For SAS SCSI command requests, I'd check the stack trace and source code for mpt_start_cmd(), mpt_scsi_start(), and mpt_scsi_init_pkt(). For command completion, I'd check mpt_intr() and mpt_scsi_destroy_pkt(). Although mpt has 287 probes (and functions), by this quick experiment we have narrowed it down to five likely probes to try first.

mpt is currently a *closed source* driver, so we've theoretically reached the end of the line using fbt—or have we? For example, to figure out the arguments for mpt_start_cmd(), we can use the mdb debugger on Solaris (which can fetch symbol information from CTF):

```
solaris# mdb -k
> mpt_start_cmd::nm -f ctype
```

continues

```
C Type
int (*)(mpt_t *, mpt_cmd_t *)

> ::print -at mpt_cmd_t
0 mpt_cmd_t {
    0 uint_t cmd_flags
    8 ddi_dma_handle_t cmd_dmahandle
[...]
    98 struct buf *cmd_ext_arq_buf
    a0 int cmd_pkt_flags
    a4 int cmd_active_timeout
    a8 struct scsi_pkt *cmd_pkt
```

The mdb example first shows the two arguments to `mpt_start_cmd()` are of type `mpt_t` and `mpt_cmd_t`. If their header files are also closed source, we can continue to use mdb to determine what the actual members are of the `mpt_cmd_t` data structure. This is shown in the previous example of `::print -at mpt_cmd_t`. Although we may not have the comments to explain what these members are for, we can DTrace their contents with known workloads, which may give us this information.

Apart from the fbt provider, the mpt driver has a collection of SDT DTrace probes inserted into the source code, available via the sdt provider. Although the sdt provider isn't officially a stable interface, it is usually much more stable than using fbt to trace kernel function calls. So, for mpt, the sdt provider may be the best place to start.

Listing the probes shows the following:

```
solaris# dtrace -ln 'sdt:mpt::'
   ID   PROVIDER         MODULE                          FUNCTION NAME
16359        sdt            mpt      mpt_ioc_task_management mpt_ioc_task_management
16360        sdt            mpt     mpt_disp_task_management mpt_disp_task_management
16361        sdt            mpt           mpt_send_inquiryVpd scsi-poll
16362        sdt            mpt                     mpt_ioctl report-phy-sata
16363        sdt            mpt                 mpt_start_cmd untagged_drain
16364        sdt            mpt               mpt_restart_hba mpt_restart_cmdioc
16365        sdt            mpt              mpt_handle_event phy-link-event
16366        sdt            mpt            mpt_handle_event event-sas-phy-link-status
16367        sdt            mpt         mpt_handle_event_sync device-status-change
16368        sdt            mpt         mpt_handle_event_sync handle-event-sync
16369        sdt            mpt           mpt_handle_hipri_dr hipri-dr
16370        sdt            mpt                 mpt_handle_dr dr
16371        sdt            mpt            mpt_check_task_mgt mpt_check_task_mgt
16372        sdt            mpt       mpt_check_scsi_io_error mpt-scsi-check
16373        sdt            mpt       mpt_check_scsi_io_error mpt_terminated
16374        sdt            mpt       mpt_check_scsi_io_error scsi-io-error
16375        sdt            mpt         mpt_process_intr io-time-on-hba-non-a-reply
16376        sdt            mpt              mpt_process_intr io-time-on-hba-a-reply
```

And, repeating the 1,000 read test:

```
solaris# dtrace -n 'sdt:mpt:: { @[probename] = count(); }'
dtrace: description 'sdt:mpt:: ' matched 18 probes
^C

  io-time-on-hba-non-a-reply                                   1007
```

Only one probe fired. Maybe there are not enough sdt probes for regular SAS I/O, but the list showed many other promising probes that our simple 1,000 read I/O test may not have triggered.

mptsassscsi.d

The `mptsasscsi.d` script counts SAS commands issued by mpt, showing the SCSI type along with details of the mpt device. This is a high-level summary script to see what SAS commands mpt is sending.

Script

To trace all mpt SAS commands, the generic `mpt_start_cmd()` function was traced and then predicated on the port type of SAS on line 36.

```
1   #!/usr/sbin/dtrace -Cs
2
3   #pragma D option quiet
4
5   /* From uts/common/sys/mpt/mpi_ioc.h */
6   #define MPI_PORTFACTS_PORTTYPE_INACTIVE         0x00
7   #define MPI_PORTFACTS_PORTTYPE_SCSI             0x01
8   #define MPI_PORTFACTS_PORTTYPE_FC               0x10
9   #define MPI_PORTFACTS_PORTTYPE_ISCSI            0x20
10  #define MPI_PORTFACTS_PORTTYPE_SAS              0x30
11
12  dtrace:::BEGIN
13  {
14          /* See /usr/include/sys/scsi/generic/commands.h for the full list. */
15          scsi_cmd[0x00] = "test_unit_ready";
16          scsi_cmd[0x08] = "read";
17          scsi_cmd[0x0a] = "write";
18          scsi_cmd[0x12] = "inquiry";
19          scsi_cmd[0x17] = "release";
20          scsi_cmd[0x1a] = "mode_sense";
21          scsi_cmd[0x1b] = "load/start/stop";
22          scsi_cmd[0x1c] = "get_diagnostic_results";
23          scsi_cmd[0x1d] = "send_diagnostic_command";
24          scsi_cmd[0x25] = "read_capacity";
25          scsi_cmd[0x28] = "read(10)";
26          scsi_cmd[0x2a] = "write(10)";
27          scsi_cmd[0x35] = "synchronize_cache";
28          scsi_cmd[0x4d] = "log_sense";
29          scsi_cmd[0x5e] = "persistent_reserve_in";
30          scsi_cmd[0xa0] = "report_luns";
31
32          printf("Tracing... Hit Ctrl-C to end.\n");
33  }
34
```

continues

```
fbt::mpt_start_cmd:entry
/args[0]->m_port_type[0] == MPI_PORTFACTS_PORTTYPE_SAS/
{
        this->mpt = args[0];
        this->mpt_name = strjoin("mpt", lltostr(this->mpt->m_instance));
        this->node_name = this->mpt->m_dip != NULL ?
            stringof(((struct dev_info *)this->mpt->m_dip)->devi_node_name) :
            "<unknown>";
        this->scsi_pkt = args[1]->cmd_pkt;
        this->code = *this->scsi_pkt->pkt_cdbp;
        this->cmd_text = scsi_cmd[this->code] != NULL ?
            scsi_cmd[this->code] : lltostr(this->code);
        @cmd[this->node_name, this->mpt_name, this->cmd_text] = count();
}

dtrace:::END
{
        printf("  %-16s %-12s %-36s %s\n", "DEVICE NODE", "MODULE", "SCSI CMD",
            "COUNT");
        printa("  %-16s %-12s %-36s %@d\n", @cmd);
}
```

Script mpt_sasscsi.d

Example

Here an application was performing synchronous writes to a ZFS file system that was using external storage (JBODs), attached via dual SAS paths (multipathing). `mptsassscsi.d` shows that two instances of mpt were handling the SCSI requests, one for each path:

```
solaris# mptsasscsi.d
Tracing... Hit Ctrl-C to end.
^C
  DEVICE NODE      MODULE       SCSI CMD                             COUNT
  pci1000,3150     mpt2         send_diagnostic_command              1
  pci1000,3150     mpt0         inquiry                              6
  pci1000,3150     mpt2         inquiry                              6
  pci1000,3150     mpt0         synchronize_cache                    26
  pci1000,3150     mpt2         synchronize_cache                    26
  pci1000,3150     mpt0         get_diagnostic_results               99
  pci1000,3150     mpt2         get_diagnostic_results               99
  pci1000,3150     mpt0         write(10)                            17299
  pci1000,3150     mpt2         write(10)                            17300
```

The counts are roughly similar for mpt0 and mpt2, showing that multipathing is balancing the load evenly.

mptevents.d

Although the previous `mptsasscsi.d` script was useful, it showed SCSI command counts for which we already had some insight at higher layers in the I/O stack. Here we trace specific mpt SAS events, excluding the transport of SCSI commands. Output is printed as it occurs, `iosnoop` style. These mpt events include performing SAS discovery on the external storage.

Script

This script makes use of the SDT probes that exist in the mpt driver, which makes extracting various details more convenient:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        /*
         * These MPI_EVENT_* definitions are from uts/common/sys/mpt/mpi_ioc.h
         */

        mpi_event[0x00000000] = "NONE";
        mpi_event[0x00000001] = "LOG_DATA";
        mpi_event[0x00000002] = "STATE_CHANGE";
        mpi_event[0x00000003] = "UNIT_ATTENTION";
        mpi_event[0x00000004] = "IOC_BUS_RESET";
        mpi_event[0x00000005] = "EXT_BUS_RESET";
        mpi_event[0x00000006] = "RESCAN";
        mpi_event[0x00000007] = "LINK_STATUS_CHANGE";
        mpi_event[0x00000008] = "LOOP_STATE_CHANGE";
        mpi_event[0x00000009] = "LOGOUT";
        mpi_event[0x0000000A] = "EVENT_CHANGE";
        mpi_event[0x0000000B] = "INTEGRATED_RAID";
        mpi_event[0x0000000C] = "SCSI_DEVICE_STATUS_CHANGE";
        mpi_event[0x0000000D] = "ON_BUS_TIMER_EXPIRED";
        mpi_event[0x0000000E] = "QUEUE_FULL";
        mpi_event[0x0000000F] = "SAS_DEVICE_STATUS_CHANGE";
        mpi_event[0x00000010] = "SAS_SES";
        mpi_event[0x00000011] = "PERSISTENT_TABLE_FULL";
        mpi_event[0x00000012] = "SAS_PHY_LINK_STATUS";
        mpi_event[0x00000013] = "SAS_DISCOVERY_ERROR";
        mpi_event[0x00000014] = "IR_RESYNC_UPDATE";
        mpi_event[0x00000015] = "IR2";
        mpi_event[0x00000016] = "SAS_DISCOVERY";
        mpi_event[0x00000017] = "SAS_BROADCAST_PRIMITIVE";
        mpi_event[0x00000018] = "SAS_INIT_DEVICE_STATUS_CHANGE";
        mpi_event[0x00000019] = "SAS_INIT_TABLE_OVERFLOW";
        mpi_event[0x0000001A] = "SAS_SMP_ERROR";
        mpi_event[0x0000001B] = "SAS_EXPANDER_STATUS_CHANGE";
        mpi_event[0x00000021] = "LOG_ENTRY_ADDED";

        sas_discovery[0x00000000] = "SAS_DSCVRY_COMPLETE";
        sas_discovery[0x00000001] = "SAS_DSCVRY_IN_PROGRESS";

        dev_stat[0x03] = "ADDED";
        dev_stat[0x04] = "NOT_RESPONDING";
        dev_stat[0x05] = "SMART_DATA";
        dev_stat[0x06] = "NO_PERSIST_ADDED";
        dev_stat[0x07] = "UNSUPPORTED";
        dev_stat[0x08] = "INTERNAL_DEVICE_RESET";
        dev_stat[0x09] = "TASK_ABORT_INTERNAL";
        dev_stat[0x0A] = "ABORT_TASK_SET_INTERNAL";
```

continues

```
        dev_stat[0x0B] = "CLEAR_TASK_SET_INTERNAL";
        dev_stat[0x0C] = "QUERY_TASK_INTERNAL";
        dev_stat[0x0D] = "ASYNC_NOTIFICATION";
        dev_stat[0x0E] = "CMPL_INTERNAL_DEV_RESET";
        dev_stat[0x0F] = "CMPL_TASK_ABORT_INTERNAL";

        printf("%-20s  %-6s %-3s    %s\n", "TIME", "MODULE", "CPU", "EVENT");
}
sdt:mpt::handle-event-sync
{
        this->mpt = (mpt_t *)arg0;
        this->mpt_name = strjoin("mpt", lltostr(this->mpt->m_instance));
        this->event_text = mpi_event[arg1] != NULL ?
            mpi_event[arg1] : lltostr(arg1);
        printf("%-20Y  %-6s %-3d -> %s\n", walltimestamp, this->mpt_name, cpu,
            this->event_text);
}
sdt:mpt::handle-event-sync
/arg1 == 0x00000016/
{
        self->mpt = (mpt_t *)arg0;
        self->discovery = 1;
}

fbt::mpt_handle_event_sync:return
/self->discovery/
{
        /* remove the PHY_BITS from the discovery status */
        this->cond = self->mpt->m_discovery & 0x0000FFFF;
        this->cond_text = sas_discovery[this->cond] != NULL ?
            sas_discovery[this->cond] : lltostr(this->cond);
        printf("%-20Y  %-6s %-3d    -> discovery status: %s\n", walltimestamp,
            this->mpt_name, cpu, this->cond_text);
        self->mpt = 0;
        self->discovery = 0;
}
sdt:mpt::device-status-change
{
        this->mpt = (mpt_t *)arg0;
        this->mpt_name = strjoin("mpt", lltostr(this->mpt->m_instance));
        this->reason = arg2;
        this->reason_text = dev_stat[this->reason] != NULL ?
            dev_stat[this->reason] : lltostr(this->reason);
        printf("%-20Y  %-6s %-3d    -> device change: %s\n", walltimestamp,
            this->mpt_name, cpu, this->reason_text);
        printf("%-20Y  %-6s %-3d       wwn=%x\n", walltimestamp,
            this->mpt_name, cpu, arg3);
}
sdt:mpt::event-sas-phy-link-status
{
        this->mpt = (mpt_t *)arg0;
        this->mpt_name = strjoin("mpt", lltostr(this->mpt->m_instance));
        this->phynum = arg1;
        printf("%-20Y  %-6s %-3d    -> phy link status, phy=%d\n",
            walltimestamp, this->mpt_name, cpu, this->phynum);
}
```

Script mptevents.d

Apart from probing all events using `sdt:mpt::handle-event-sync`, some events print an extra line or two of details by probing them separately. To differentiate their output, their event details are indented by three more spaces. More such probes from sdt or fbt could be added to print extra details on events, if required.

The CPU ID is printed only as a reminder that the output may be a little shuffled because of the way dtrace collects data from its per-CPU switch buffers for printing; if this potential for shuffling was a problem, a time stamp field (in nanoseconds) could be printed for postsorting.

Example

While an application was busy writing to external storage on this system, there were no specific mpt events occurring. To trigger some, we removed a disk from a JBOD:

```
solaris# mptevents.d
TIME                      MODULE CPU    EVENT
2009 Dec 31 09:02:31  mpt0   1   -> SAS_DISCOVERY
2009 Dec 31 09:02:31  mpt0   1      -> discovery status: SAS_DSCVRY_IN_PROGRESS
2009 Dec 31 09:02:31  mpt0   1   -> SAS_PHY_LINK_STATUS
2009 Dec 31 09:02:31  mpt0   12     -> phy link status, phy=13
2009 Dec 31 09:02:31  mpt0   1   -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:31  mpt0   1      -> device change: INTERNAL_DEVICE_RESET
2009 Dec 31 09:02:31  mpt0   1         wwn=500163600004db49
2009 Dec 31 09:02:31  mpt0   1   -> SAS_DISCOVERY
2009 Dec 31 09:02:31  mpt0   1      -> discovery status: SAS_DSCVRY_COMPLETE
2009 Dec 31 09:02:31  mpt2   7      -> phy link status, phy=13
2009 Dec 31 09:02:31  mpt2   15  -> SAS_DISCOVERY
2009 Dec 31 09:02:31  mpt2   15     -> discovery status: SAS_DSCVRY_IN_PROGRESS
2009 Dec 31 09:02:31  mpt2   15  -> SAS_PHY_LINK_STATUS
2009 Dec 31 09:02:31  mpt2   15  -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:31  mpt2   15     -> device change: INTERNAL_DEVICE_RESET
2009 Dec 31 09:02:31  mpt2   15        wwn=500163600015c7c9
2009 Dec 31 09:02:31  mpt2   15  -> SAS_DISCOVERY
2009 Dec 31 09:02:31  mpt2   15     -> discovery status: SAS_DSCVRY_COMPLETE
2009 Dec 31 09:02:31  mpt0   1   -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:31  mpt0   1      -> device change: CMPL_INTERNAL_DEV_RESET
2009 Dec 31 09:02:31  mpt0   1         wwn=500163600004db49
2009 Dec 31 09:02:31  mpt2   15  -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:31  mpt2   15     -> device change: CMPL_INTERNAL_DEV_RESET
2009 Dec 31 09:02:31  mpt2   15        wwn=500163600015c7c9
2009 Dec 31 09:02:32  mpt2   15  -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:32  mpt2   15     -> device change: NOT_RESPONDING
2009 Dec 31 09:02:32  mpt2   15        wwn=500163600015c7c9
2009 Dec 31 09:02:33  mpt0   1   -> SAS_DEVICE_STATUS_CHANGE
2009 Dec 31 09:02:33  mpt0   1      -> device change: NOT_RESPONDING
2009 Dec 31 09:02:33  mpt0   1         wwn=500163600004db49
^C
```

The output shows SAS_DEVICE_STATUS_CHANGE events in response to pulling a disk; the reason code for the status change begins as INTERNAL_DEVICE_RESET and finishes as NOT_RESPONDING.

mptlatency.d

Finally, this section presents a short script to show latency by mpt SCSI command.

Script

Time stamps are already kept in the `mpt_cmd_t` such that they can be read in the `sdt:mpt::io-time-on-hba-non-a-reply` probe to calculate command time.

```
#!/usr/sbin/dtrace -s

dtrace:::BEGIN
{
        /* See /usr/include/sys/scsi/generic/commands.h for the full list. */
        scsi_cmd[0x00] = "test_unit_ready";
        scsi_cmd[0x08] = "read";
        scsi_cmd[0x0a] = "write";
        scsi_cmd[0x12] = "inquiry";
        scsi_cmd[0x17] = "release";
        scsi_cmd[0x1a] = "mode_sense";
        scsi_cmd[0x1b] = "load/start/stop";
        scsi_cmd[0x1c] = "get_diagnostic_results";
        scsi_cmd[0x1d] = "send_diagnostic_command";
        scsi_cmd[0x25] = "read_capacity";
        scsi_cmd[0x28] = "read(10)";
        scsi_cmd[0x2a] = "write(10)";
        scsi_cmd[0x35] = "synchronize_cache";
        scsi_cmd[0x4d] = "log_sense";
        scsi_cmd[0x5e] = "persistent_reserve_in";
        scsi_cmd[0xa0] = "report_luns";
}

sdt:mpt::io-time-on-hba-non-a-reply
{
        this->mpt = (mpt_t *)arg0;
        this->mpt_cmd = (mpt_cmd_t *)arg1;

        this->mpt_name = strjoin("mpt", lltostr(this->mpt->m_instance));
        this->delta = (this->mpt_cmd->cmd_io_done_time -
            this->mpt_cmd->cmd_io_start_time) / 1000;
        this->code = *this->mpt_cmd->cmd_cdb;
        this->cmd_text = scsi_cmd[this->code] != NULL ?
            scsi_cmd[this->code] : lltostr(this->code);
        @[this->mpt_name, this->cmd_text] = quantize(this->delta);
}

dtrace:::END
{
        printf("Command Latency (us):\n");
        printa(@);
}
```

Script mptlatency.d

Example

This example has been trimmed to show just one distribution plot from the many that were printed:

```
solaris# mptlatency.d
dtrace: script '/chapters/disk/sas/mptlatency.d' matched 3 probes
^C
```

```
CPU     ID                    FUNCTION:NAME
  8      2                            :END Command Latency (us):

  mpt0                                              read(10)
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@@                  277
            1024 |@@@@@@@@@@@@@                            161
            2048 |@@@@@                                    61
            4096 |                                         0
            8192 |                                         1
           16384 |                                         0
```

The system was performing disk reads, whose latency as seen by mpt is shown in a distribution plot. The latency values look very good, with a large distribution in the 512-microsecond to 2-millisecond range.

Case Studies

In this section, we show applying some of the one-liners, scripts, and methods discussed in this chapter to specific instances of disk I/O analysis.

Shouting in the Data Center: A Personal Case Study (Brendan)

In December 2008, I used a DTrace-based tool called *Analytics* (see Chapter 14, Analytics) to discover that shouting at disk arrays can cause significant disk I/O latency. Bryan Cantrill (coinventor of DTrace) immediately filmed me doing this and posted it online.[18] Here I'll show you how to measure the same effect using DTrace scripting, should analytics not be available on your system.

The problem arises when vibrations (or shock) cause high I/O latency on mechanical disk drives with rotating disk platters and seeking heads. If a disk head moves slightly off-track during read, it may read incorrect data, fail the disk sector CRC, and retry the read (perhaps by automatically repositioning the head to slightly different locations in an attempt to realign); this may happen a number of times before the disk successfully reads the data, causing higher-than-usual latency. For writes, disks may be much more careful when detecting head misalignment in order to prevent writing to the wrong location and corrupting data.

In the *Shouting in the Datacenter* video, I was testing maximum write performance of two JBODs (arrays of disks) when I noticed that the overall throughput was lower than expected. I investigated using DTrace-based analytics and discovered

18. This is available at *www.youtube.com/watch?v=tDacjrSCeq4*, where it has had more than 650,000 views, making it the most-watched video about Sun in the history of Sun Microsystems.

that a single disk had high latency. I began to suspect vibration was the cause, because it was missing a screw on the drive bracket and was loose. Then it began to behave normally (same latency as other disks). Wanting to re-create the vibration issue, I hit upon the idea of shouting at it. That's when I discovered that my shout affected *every* disk in the array. Disks just don't like being shouted at.

To detect this with DTrace, we need an effective way to identify occasional high latency. Metrics such as average disk service time won't do it: My disks were performing thousands of disk I/Os, and a few slow ones would simply get lost in the average. There is where `quantize()` and `lquantize()` come in handy.

I could DTrace this from any layer of the I/O stack, but I decided to stick to the block device layer: It has the stable io provider. I'll describe in the following sections how I wrote a custom script to show only the high latencies by disk device, and I'll also show how well the other scripts in this chapter identify the issue.

Workload

I had some unused SATA system disks that I could write over using the `dd(1)` command:

```
# dd if=/dev/zero of=/dev/rdsk/c0t0d0s0 bs=64k
```

> **Warning**
>
> If you run this on a disk, it will erase all data!

While this write workload was running, I used `iolatency.d` to get an accurate average I/O time:

```
solaris# iolatency.d
Tracing... Hit Ctrl-C to end.
^C

  TYPE              PAGEIO     RESULT       COUNT      AVG(us)     TOTAL(ms)
  phys-write            no    Success        1478         3588          5303
```

The average IO time was 3.6 ms.

Perturbation

To perturb this workload, I shouted at the disk—twice—from a couple of inches. First I shouted very loudly and then even louder—as loud as I possibly could. I wish I could quantify *how* loud, but last time I used a decibel meter, I reached its max. From that close, a shout can be so loud that I've heard it described as a percussive shock wave rather than sound vibration.

Observation

During the shouts, I ran a few DTrace scripts from this chapter, plus one customized for this situation. `iolatency.d` isn't suited for detecting this, because it gives latency results in terms of averages, which can hide the severity of the slowest I/O.

```
solaris# iolatency.d

Tracing... Hit Ctrl-C to end.
^C
  TYPE                PAGEIO      RESULT        COUNT      AVG(us)     TOTAL(ms)
  phys-read               no     Success           18        10751           193
  phys-write              no     Success          536        17227          9233
```

The average I/O latency has increased from 3.6 ms to 17.2 ms, almost five times worse, so we have reason to be suspicious and investigate further.

`rwtime.d` shows read/write I/O latency:

```
solaris# rwtime.d
Tracing... Hit Ctrl-C to end.
^C
   read I/O, us

           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@                                     2
            4096 |@@@@@@@@@@@@@                            6
            8192 |@@@@@@@@@@@@@                            6
           16384 |@@@@@@@@@                                4
           32768 |                                         0

   write I/O, us

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@                                 96
            1024 |@@@@@                                    58
            2048 |@                                        7
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@                       216
           16384 |@@@@@@                                   73
           32768 |@                                        16
           65536 |                                         2
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         2
         2097152 |                                         0


  average read I/O, us                                          10793
  average write I/O, us                                         19236
```

This effectively shows the problem with average latency: Although the average measured here was 19.2 ms (at a slightly different interval than before), the distribution plot for writes showed the outliers clearly—two writes in the one- to two-*second* range.

The `disklatency.d` script can be used to identify the disk affected:

```
solaris# disklatency.d
Tracing... Hit Ctrl-C to end.
[...]

   sd0 (227,0), us:

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@                               176
            1024 |@@@@@@@@@                                148
            2048 |@                                        20
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@                           234
           16384 |@@@@@                                    81
           32768 |@                                        16
           65536 |                                         2
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         2
         2097152 |                                         0
```

The data shows that sd0 was the affected disk.

Finally, `shoutdetector.d` is a custom script that shows only slow I/Os—those longer than 500 ms. Unlike the previous scripts, which ran until Ctrl-C was hit, this script prints output every second:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
inline int SLOWIO = 500;                        /* millisecond threshould */

io:::start
{
        start_time[arg0] = timestamp;
}

io:::done
/(this->start = start_time[arg0]) &&
    (this->delta_ms = (timestamp - this->start) / 1000000) &&
    (this->delta_ms > SLOWIO)/
{
        @[args[1]->dev_statname] = lquantize(this->delta_ms, 0, 5000, 100);
        start_time[arg0] = 0;
}

profile:::tick-1sec
{
        printf("%Y:", walltimestamp);
```

```
            printa(@);
            trunc(@);
}

Script shoutdetector.d
```

Here's the output:

```
# shoutdetector.d
2010 Jan  1 22:54:16:
2010 Jan  1 22:54:17:
2010 Jan  1 22:54:18:
2010 Jan  1 22:54:19:
2010 Jan  1 22:54:20:
2010 Jan  1 22:54:21:
2010 Jan  1 22:54:22:
  sd0
           value  ------------- Distribution ------------- count
             900 |                                         0
            1000 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
            1100 |                                         0

2010 Jan  1 22:54:23:
2010 Jan  1 22:54:24:
2010 Jan  1 22:54:25:
  sd0
           value  ------------- Distribution ------------- count
            1500 |                                         0
            1600 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
            1700 |                                         0

2010 Jan  1 22:54:26:
2010 Jan  1 22:54:27:
2010 Jan  1 22:54:28:
2010 Jan  1 22:54:29:
2010 Jan  1 22:54:30:
2010 Jan  1 22:54:31:
2010 Jan  1 22:54:32:
2010 Jan  1 22:54:33:
2010 Jan  1 22:54:34:
2010 Jan  1 22:54:35:
2010 Jan  1 22:54:36:
```

Perfect—it's showing only the data I'm interested in (the extremely slow I/O), along with the disk name. This also shows that my second shout really was louder than my first: The first caused a 1.0 second I/O, and the second caused a 1.6 second I/O.

DTracing an Unfamiliar I/O Driver (SATA)

Although I had DTraced a wide variety of subsystems and applications before, I had never used DTrace on SATA before writing this chapter. I had also never seen the SATA source code nor knew how the internals worked. In less than a day, I had figured this all out *and* had written most of the scripts included in the "SATA" section. The secret isn't superhuman powers (or superstrong coffee) but rather an effective strategy for approaching unfamiliar drivers or subsystems and gaining quick familiarity with them. In the following sections, I've documented exactly how I did it, as a case study for DTracing the unknown. This isn't the only way to do this, but it is one effective strategy that may serve you well.

Documentation

I started by searching for some high-level documentation for SATA to get a basic understanding of its purpose and terminology. I also looked for functional diagrams to understand how it fits into the bigger picture. The functional diagram in the "SATA Scripts" section is exactly what I was looking for, but I couldn't find anything like it at the time.

I checked the following sources for driver documentation:

- Man pages (kernel drivers often have them)
- Internet search engines
- *Solaris Internals* (McDougall and Mauro, 2006)
- Comments in driver source code

I reached for the source code, not to read the code itself but to look for the large descriptive block comments, often found at the top of source files, which sometimes include functional diagrams.

I was eventually able to figure out the functional diagram for the SATA drivers, as shown in Figure 4-9.

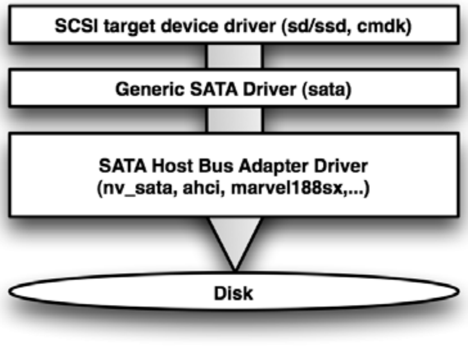


Figure 4-9 SATA stack

Stable Providers

I checked whether DTrace had a stable provider for SATA yet; it doesn't. You can check for stable providers in the "Providers" section of the DTrace Guide.[19]

If a stable provider exists, the provider documentation will likely contain example scripts that can be run immediately, which would work on any OS that had that provider.

Unstable Providers: sdt

Without a stable provider, I next checked the sdt (statically defined tracing) provider. If any sdt probes exist, they are likely to be interesting, because someone would have added them specifically for use by DTrace—they may also be reasonably stable. Engineers try not to change sdt probes, because that would break DTrace scripts based on them, but will if they need to[20] (and are allowed to, since it isn't a committed stable provider).

I knew that these kernel drivers are called *sata* and *nv_sata*, so I could specify that as the module name and attempt to list probes from the sdt provider:

```
solaris# dtrace -ln 'sdt:sata::,sdt:nv_sata::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
dtrace: failed to match sdt:sata::: No probe matches description
30987        sdt           nv_sata                      nv_sgp_error sgpio-error
30988        sdt           nv_sata                     nv_sgp_locate sgpio-locate
30989        sdt           nv_sata               nv_sgp_drive_active sgpio-active
30990        sdt           nv_sata           nv_sgp_activity_led_ctl sgpio-new-led-
state
30991        sdt           nv_sata           nv_sgp_activity_led_ctl sgpio-activity-
state
30992        sdt           nv_sata                       nv_sgp_init sgpio-cmd
```

The `grep` search command works fine too:

```
solaris# dtrace -ln 'sdt:::' | grep sata
36362        sdt               mpt                         mpt_ioctl report-phy-sata
30987        sdt           nv_sata                      nv_sgp_error sgpio-error
30988        sdt           nv_sata                     nv_sgp_locate sgpio-locate
30989        sdt           nv_sata               nv_sgp_drive_active sgpio-active
30990        sdt           nv_sata           nv_sgp_activity_led_ctl sgpio-new-led-
state
30991        sdt           nv_sata           nv_sgp_activity_led_ctl sgpio-activity-
state
30992        sdt           nv_sata                       nv_sgp_init sgpio-cmd
```

This picked up an extra probe from the mpt driver.

19. This is currently at *http://wikis.sun.com/display/DTrace/Providers*.

20. I recently changed `sdt:::arc-miss` for CR 6876733: "sdt:::arc-hit and sdt:::arc-miss provide inconsistent args[0]."

So, there are sdt probes available, but only for the nv_sata driver, not sata. I'd rather write scripts based on the generic sata driver, because they are more likely to work elsewhere (systems with SATA HBAs that aren't nv_sata). But I decided to check out these sdt probes first; they were probably more stable.

Frequency Count sdt

To quickly get familiar with these sdt probes, I started by applying a simple known workload to see what probes would fire. I used the `dd(1)` command to perform 10,000 reads from a SATA disk:

```
solaris# dd if=/dev/rdsk/c3t0d0s0 of=/dev/null bs=8k count=10000
1000+0 records in
1000+0 records out
```

While using DTrace to frequency count which sdt probes fired:

```
solaris# dtrace -n 'sdt:nv_sata:: { @[probename] = count(); }'
dtrace: description 'sdt:nv_sata:: ' matched 6 probes
^C

  sgpio-activity-state                                              164
  sgpio-new-led-state                                               164
  sgpio-active                                                    10017
```

I hoped to find a probe corresponding to the known workload of 10,000 reads. This showed that the `sgpio-active` probe fired 10,017 times. `sgpio-active`? That name didn't seem promising, but perhaps it could be used to trace I/O anyway.

I knew I was unlikely to find any documentation for this SDT probe outside of the kernel source code. To find where it lives in the source, I read the `FUNCTION` field in the previous `dtrace -l` outputs, which showed that it is in the `nv_sgp_drive_active()` function.

This function is in `uts/common/io/sata/adapters/nv_sata/nv_sata.c` and is as follows:

```
/*
 * nv_sgp_drive_active
 * Sets the flag used to indicate that the drive has been accessed and the
 * LED should be flicked off, then on.  It is cleared at a fixed time
 * interval by the LED taskq and set by the sata command start.
 */
static void
nv_sgp_drive_active(nv_ctl_t *nvc, int drive)
{
        nv_sgp_cmn_t *cmn;

```

```
6959         if (nv_sgp_check_set_cmn(nvc) == NV_FAILURE)
6960                 return;
6961         cmn = nvc->nvc_sgp_cmn;
6962
6963         DTRACE_PROBE1(sgpio__active, int, drive);
6964
6965         mutex_enter(&cmn->nvs_slock);
6966         cmn->nvs_activity |= (1 << drive);
6967         mutex_exit(&cmn->nvs_slock);
6968 }
```

The description shows that this function controls the blinking of the drive LED. The DTrace probe is on line 6963 and shows the arguments it provides: `drive`, which is of type `int`. This was not what I was after. Had it been a pointer to a structure, there might have been many useful members to read that might help me identify the SATA command, SCSI command, HBA details, and so on. There wasn't much I could do with this one integer, apart from the following:

```
# dtrace -qn 'sgpio-active { printf("drive %d just blinked its LED!\n", arg0); }'
drive 2 just blinked its LED!
drive 2 just blinked its LED!
drive 2 just blinked its LED!
drive 2 just blinked its LED!
^C
```

So, although I couldn't use sdt to trace read I/O (remember, nothing other than the LED probe fired ~10,000 times during my experiment), perhaps one of the other probes would be useful, for example for tracing error events. After reading the comments above these functions, I found that most of the others are also for tracing LED activity (`sgpio-error` is for the error LED!). And I never saw `sgpio-cmd` fire, even when physically swapping SATA disks.

Unstable Providers: fbt

The fbt (function boundary tracing) provider traces entry and returns from kernel functions. This checks what function entry probes fbt can see for the sata and nv_sata drivers:

```
solaris# dtrace -ln 'fbt:sata::entry,fbt:nv_sata::entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
65557        fbt              sata              sata_trace_rbuf_alloc entry
65559        fbt              sata               sata_trace_rbuf_free entry
65561        fbt              sata         sata_validate_sata_hba_tran entry
65563        fbt              sata                 sata_scsi_tgt_init entry
65565        fbt              sata                sata_scsi_tgt_probe entry
65567        fbt              sata                 sata_scsi_tgt_free entry
65569        fbt              sata                    sata_scsi_start entry
65571        fbt              sata                    sata_scsi_reset entry
65573        fbt              sata                    sata_scsi_abort entry
```

continues

```
65575       fbt             sata                    sata_scsi_getcap entry
65577       fbt             sata                    sata_scsi_setcap entry
[...]
solaris# dtrace -ln 'fbt:sata::entry,fbt:nv_sata::entry' | wc -l
     260
```

That's 259 probes, representing 259 kernel functions that make up the sata and nv_sata providers. Some of these functions would likely fire for the events I was interested in: read/write I/O, other SATA commands, SCSI commands, and so on. Some of the function arguments might also point to interesting data, such as SATA command types, SATA command errors, I/O sizes, and device details. However, I didn't know the functions or the arguments, and they weren't likely to be documented beyond the source code. And I couldn't find any examples of using fbt to DTrace SATA on the Internet. I was on my own.

I could bring up the SATA code and start reading through it, to know what to use fbt with. But I didn't even know where to start reading. The SATA code is more than 20,000 lines, and the nv_sata code is 7,000 lines, which could take days to read through. It's often quicker to use DTrace to see which probes fire and start with those functions.

Frequency Count fbt

My experiment was a known workload of 10,000 reads to a SATA disk, as I did previously with the sdt provider. I would also physically swap some disks to trigger other SATA events.

The output was as follows:

```
solaris# dtrace -n 'fbt:sata::entry,fbt:nv_sata::entry
{ @[probemod, probefunc] = count(); }
    END { printa("   %-10s %-40s %@10d\n", @); }'
dtrace: description 'fbt:sata::entry,fbt:nv_sata::entry ' matched 260 probes

CPU     ID                    FUNCTION:NAME
 11      2                             :END
   sata       sata_set_cache_mode                               1
   nv_sata    nv_abort_active                                   2
   nv_sata    nv_monitor_reset                                  2
[...truncated...]
   nv_sata    nv_start_nodata                                 125
   sata       sata_txlt_write                                 189
   nv_sata    nv_sgp_csr_read                                 574
   nv_sata    nv_sgp_csr_write                                574
   nv_sata    nv_sgp_write_data                               574
   sata       sata_txlt_read                                10009
   nv_sata    nv_bm_status_clear                            10185
   nv_sata    nv_intr_dma                                   10185
   nv_sata    nv_start_dma                                  10185
   nv_sata    nv_start_dma_engine                           10185
   sata       sata_txlt_rw_completion                       10185
   sata       sata_dma_buf_setup                            10218
   sata       sata_adjust_dma_attr                          10220
```

```
nv_sata    nv_start_async                              10302
nv_sata    mcp5x_packet_complete_intr                  10314
nv_sata    nv_complete_io                              10314
nv_sata    nv_program_taskfile_regs                    10316
nv_sata    nv_sgp_drive_active                         10316
nv_sata    nv_start_common                             10316
nv_sata    nv_wait                                     10316
sata       sata_hba_start                              10320
nv_sata    nv_copy_registers                           10324
sata       sata_scsi_destroy_pkt                       10328
nv_sata    nv_sata_start                               10329
sata       sata_scsi_init_pkt                          10330
sata       sata_pkt_free                               10337
sata       sata_pkt_alloc                              10339
sata       sata_txlt_generic_pkt_info                  10339
sata       sata_scsi_start                             10341
sata       sata_common_free_dma_rsrcs                  10363
sata       sata_validate_sata_address                  10428
sata       sata_validate_scsi_address                  10428
nv_sata    mcp5x_intr                                  27005
nv_sata    nv_read_signature                           27437
nv_sata    nv_sgp_drive_disconnect                     27439
nv_sata    nv_sgp_check_set_cmn                        37763
sata       sata_get_device_info                        51551
nv_sata    mcp5x_intr_port                             54009
```

I started by identifying the I/O probes, trying to identify the highest-level probes first. For SATA, this would be a probe (or probes) to trace the issuing of SATA commands, and probes for the response.

Since I assumed[21] that I had issued 10,000 disk I/Os via SATA, I started by looking at probes that fired around 10,000 times. Twenty-seven such probes are listed, some of which look promising: Their names are as generic as possible (for example, `sata_start` and `sata_done`). For command start, there is `nv_start_common`, `sata_hba_start`, and `nv_sata_start`. For a command completion probe, there is `nv_complete_io` but nothing obvious in the SATA layer.

The possibilities I've found are as follows:

SATA command start: `nv_start_common()`, `sata_hba_start()`, and `nv_sata_start()`

SATA command completion: `nv_complete_io()`

21. While I used `dd` to cause 10,000 read I/Os on the `/dev/rdsk` path, I don't know for certain that these were still 10,000 read I/Os by the time SATA issued them. SATA could split I/Os into smaller sizes to overcome hardware bus or buffer limitations. It could also be clever and aggregate I/Os into larger sizes. This is why I chose 10,000 as a count and not, say, 100. If my 10,000 events are doubled or halved or whatever, they are still likely to amount to a large count that should stand out above the noise.

Examine Stack Backtraces: I/O Start

To understand the relationships among these functions, I could examine stack backtraces. Since I wanted to capture as much of the stack backtrace as possible, for the start probe I started at the nv_sata layer—its stack backtrace should include the sata layer:

```
solaris# dtrace -x stackframes=100 -n
'fbt::nv_start_common:entry,fbt::nv_sata_start:entry
{ @[probefunc, stack()] = count(); }'
[...]
  nv_start_common
              nv_sata`nv_start_async+0x74
              nv_sata`nv_sata_start+0x132
              sata`sata_hba_start+0x112
              sata`sata_txlt_read+0x412
              sata`sata_scsi_start+0x38b
              scsi`scsi_transport+0xb5
              sd`sd_start_cmds+0x2e8
              sd`sd_core_iostart+0x186
              sd`sd_mapblockaddr_iostart+0x306
              sd`sd_xbuf_strategy+0x50
              sd`xbuf_iostart+0x1e5
              sd`ddi_xbuf_qstrategy+0xd3
              sd`sdstrategy+0x101
              genunix`default_physio+0x3cb
              genunix`physio+0x25
              sd`sdread+0x16b
              genunix`cdev_read+0x3d
              specfs`spec_read+0x233
              genunix`fop_read+0xa7
              genunix`read+0x2b8
              genunix`read32+0x22
              unix`sys_syscall32+0x101
              100
```

The stack included here was the longest visible stack backtrace, showing that `nv_start_common()` is deep in the code path. I could also see that `sata_hba_start()` calls `nv_sata_start()`, which calls another function and then `nv_start_common()`.

The point of tracing at nv_sata shown earlier was simply to get a long illustrative stack trace, which I succeeded in doing. Now I wanted to pick a function to actually trace, which would preferably be in the generic sata driver and not specific to nv_sata (Nvidia SATA HBA). Examining the other stack frames showed that both `sata_hba_start()` and `sata_scsi_start()` were common to all stacks and in the sata layer.

Examine Stack Backtraces: I/O Done

For the I/O completion probe, I used a different strategy. I didn't want to show stack traces from nv_sata, since that would show the shortest, not the longest, stack this time:

```
solaris# dtrace -n 'fbt::nv_complete_io:entry { @[stack()] = count(); }'
dtrace: description 'fbt::nv_complete_io:entry ' matched 1 probe
^C


              nv_sata`mcp5x_packet_complete_intr+0xbd
              nv_sata`mcp5x_intr_port+0x8a
              nv_sata`mcp5x_intr+0x45
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
            10005
```

The stack backtrace can show only where a thread has been, and on the return path it is beginning from nv_sata, so there isn't much stack backtrace to see yet. I need to trace from sata or higher.

Back in step 6, I never found anything obvious from the sata driver for the completion events. I could try a couple of other ways to find it.

Experimentation

I expected a completion event to have `nv_complete_io()` in its stack trace and be fired around 10,000 times for my experiment. I frequency counted all functions and stack traces from the sata layer and started by eyeballing the output for something containing `nv_complete_io()` in its stack:

```
solaris# dtrace -x stackframes=100 -n 'fbt:sata::entry
{ @[probefunc, stack()] = count(); }'
dtrace: description 'fbt:sata::entry ' matched 179 probes
^C
[...]
  sata_pkt_free
              sata`sata_scsi_destroy_pkt+0x3f
              scsi`scsi_destroy_pkt+0x21
              sd`sd_destroypkt_for_buf+0x20
              sd`sd_return_command+0x1c3
              sd`sdintr+0x58d
              scsi`scsi_hba_pkt_comp+0x15c
              sata`sata_txlt_rw_completion+0x1d3
              nv_sata`nv_complete_io+0x7f
              nv_sata`mcp5x_packet_complete_intr+0xbd
              nv_sata`mcp5x_intr_port+0x8a
              nv_sata`mcp5x_intr+0x45
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
            10000
[...]
  sata_txlt_rw_completion
              nv_sata`nv_complete_io+0x7f
              nv_sata`mcp5x_packet_complete_intr+0xbd
              nv_sata`mcp5x_intr_port+0x8a
              nv_sata`mcp5x_intr+0x45
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
            10003
```

I truncated the output down to the two most interesting stacks for probes that fired around 10,000 times. I looked for stacks containing nv_sata in the stack, which identifies the I/O completion path. I also looked for generic function names.

`sata_txlt_rw_completion()` looked like it would work, but perhaps only for SATA read/write commands (is that what the `rw` means? I could check the source code to find out). I wanted to identify a return probe for all SATA commands, whether read/write I/O or other type. `sata_pkt_free()`, which frees the sata packet, might work for everything. This sounded better: It would happen near the end of the completion code path (after processing of the packet is done) and should match both read/write and other commands. A drawback is that it might catch packets that were never actually sent but errored before reaching nv_sata and the packets themselves were freed.

Source Code

Another way to see how we return from the sata driver to nv_sata is to start reading the source code from the `nv_complete_io()` function to see how it gets there.

From `uts/common/io/sata/adapters/nv_sata/nv_sata.c`, here's the output:

```
3567 static void
3568 nv_complete_io(nv_port_t *nvp, sata_pkt_t *spkt, int slot)
3569 {
[...]
3598         if (spkt->satapkt_comp != NULL) {
3599                 mutex_exit(&nvp->nvp_mutex);
3600                 (*spkt->satapkt_comp)(spkt);
3601                 mutex_enter(&nvp->nvp_mutex);
3602         }
```

The function call is on line 3600, which is C syntax for calling a function from a "function pointer." These make reading the code very difficult, since on line 3600 we can't read what function was called but only the variable `satapkt_comp` that contained the function. I could search the code to see where the `satapkt_comp` function was set. This found only two possibilities:

```
spx->txlt_sata_pkt->satapkt_comp = sata_txlt_rw_completion;

spx->txlt_sata_pkt->satapkt_comp = sata_txlt_atapi_completion;
```

I'd already found the first one by experimentation, but I hadn't yet discovered `sata_txlt_atapi_completion()`. I could stop right there and trace SATA completion events by tracing both of those functions, provided their arguments were useful enough. Or I could keep reading through the code to see whether all paths led to something more generic so that I could dtrace everything with just one probe—instead of two. I already suspected from experimentation that all roads eventually led to `sata_pkt_free()`.

Now I had two possibilities for DTracing the completion of SATA commands:

- Trace both `sata_txlt_rw_completion()` and `sata_txlt_atapi_completion()`
- Trace `sata_pkt_free()`, and be careful not to count packets that were never sent

Their arguments might provide the tie-breaker.

Examine Function Arguments

Back to the SATA command start probe. So far, I'd found two possible functions to pick from in sata: `sata_scsi_start()` and `sata_hba_start()`. I turned to the source code of the sata driver to see which had the best information in its arguments, which are DTraceable as `args[]`:

```
static int sata_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
static int sata_hba_start(sata_pkt_txlate_t *spx, int *rval)
```

Interesting. Since `sata_scsi_start()` is highest in the stack backtrace, it is executed first in the sata driver and doesn't have any sata information yet in its arguments, only scsi information. (If I wanted to trace scsi information, I'd usually move further up the stack into scsi or sd.) `sata_hba_start()` is called deeper and has `sata_pkt_txlate_t`, which looks more promising. It is defined in `uts/common/sys/sata/impl/sata.h` as:

```
typedef struct sata_pkt_txlate {
        struct sata_hba_inst    *txlt_sata_hba_inst;
        struct scsi_pkt         *txlt_scsi_pkt;
        struct sata_pkt         *txlt_sata_pkt;
        ddi_dma_handle_t        txlt_buf_dma_handle;
        uint_t                  txlt_flags;     /* data-in / data-out */
        uint_t                  txlt_num_dma_win; /* number of DMA windows */
        uint_t                  txlt_cur_dma_win; /* current DMA window */
[...]
```

Excellent—the top three members to this function looked as if they would provide HBA instance information, SCSI packet information, and SATA packet information, which is everything I really wanted. I read the definitions for those data types and found the following:

```
struct sata_pkt {
        int             satapkt_rev;            /* version */
        struct sata_device satapkt_device;      /* Device address/type */

                                                /* HBA driver private data */
        void            *satapkt_hba_driver_private;

                                                /* SATA framework priv data */
        void            *satapkt_framework_private;

                                                /* Rqsted mode of operation */
        uint32_t        satapkt_op_mode;

        struct sata_cmd satapkt_cmd;            /* composite sata command */
        int             satapkt_time;           /* time allotted to command */
        void            (*satapkt_comp)(struct sata_pkt *); /* callback */
        int             satapkt_reason;         /* completion reason */
};
```

The two members that looked most interesting there were `satapkt_cmd` for the SATA command itself and `satapkt_reason` for the reason the command returned (success or error).

I could search the source code to see how these arguments are processed and what they mean. But first I performed a quick experiment with a known workload to see how they looked. I started with `satapkt_reason`, because it's just an `int`, and "completion reason" sounded like it could be interesting—it might tell me why commands completed (success or error):

```
solaris# dtrace -n 'fbt::sata_hba_start:entry
{ @[args[0]->txlt_sata_pkt->satapkt_reason] = count(); }'
dtrace: description 'fbt::sata_hba_start:entry ' matched 1 probe
^C

        2                1
        0            10026
```

So, `satapkt_reason` of 0 occurred 10,026 times, and 2 occurred once. I didn't know what these codes meant, but I could search the sata driver source to see how it processes them. Searching for the text `satapkt_reason`, I found examples like this:

```
6534         if (sata_pkt->satapkt_reason == SATA_PKT_COMPLETED) {
6535                 /* Normal completion */
```

This is from `uts/common/io/sata/impl/sata.c`. On line 6534, we can see that `satapkt_reason` is compared with the constant `SATA_PKT_COMPLETED`. Searching for that constant finds it defined in `/usr/include/sys/sata/sata_hba.h`:

```
#define SATA_PKT_BUSY                      -1      /* Not completed, busy */
#define SATA_PKT_COMPLETED                 0       /* No error */
#define SATA_PKT_DEV_ERROR                 1       /* Device reported error */
#define SATA_PKT_QUEUE_FULL                2       /* Not accepted, queue full */
#define SATA_PKT_PORT_ERROR                3       /* Not completed, port error */
[...]
```

These defines can serve as a lookup table of codes to descriptions. Here we can see that 0 means “No error,” and 2 means “Not accepted, queue full.” This list also confirmed my suspicion about the `satapkt_reason` code—it *is* interesting.

DTrace has access to the integer code; I’d rather it printed out a descriptive string. The previous table could be converted into an associative array to be used in DTrace programs for translation, or I could convert it to a translator file. To try the associative array approach, I saved the defines into a file called `reasons.h` and converted it using sed:[22]

```
# sed 's/[^0-9-]*/    sata_reason[/;s: */\* :] = ":;s: ..$:";:' reasons.h
        sata_reason[-1] = "Not completed, busy";
        sata_reason[0] = "No error";
        sata_reason[1] = "Device reported error";
        sata_reason[2] = "Not accepted, queue full";
        sata_reason[3] = "Not completed, port error";
[...]
```

The script `satareasons.d` demonstrates using such a translation table to print out command completion reasons.

`satapkt_reason` was just an integer. Another interesting member would be the SATA command itself, `satapkt_cmd`, but that is a struct that turned out to be somewhat complex to unravel. Remember, the driver code processes it, so I do have examples to pick through.

Latency

I/O latency is extremely important for performance analysis, but it’s rare to find I/O latency precalculated and ready to be fetched from a function argument. I often need to calculate this myself, by tracing two events (`start` and `done`) and calculating the delta (`done_time – start_time`). The trick is associating the events together in one DTrace action block so that I have both times available to perform that calculation.

22. It can be handy to know sed programming for times like this, but it’s really not necessary; the same conversion could have been done by hand in a text editor in a minute or two.

I can't use a global variable to store the start time, because many I/Os may be in-flight concurrently and can't share the same variable. Thread-local variables can't be used either: The completion event is usually on a different thread than the start event. Associative arrays *can* be used as long as I can find a unique key to store the start time against, a key that is available to both the `start` and `done` functions.

For the `start` function, one option was `sata_hba_start()`:

```
static int sata_hba_start(sata_pkt_txlate_t *spx, int *rval) ;
```

And for completion, here's `sata_pkt_free()`:

```
static void sata_pkt_free(sata_pkt_txlate_t *spx) ;
```

Ideally, an argument would be something like "unique ID for this I/O command," which I'd pull from the `args[]` array and use as a key in an associative array. I didn't see it. They both do have `sata_pkt_txlate_t` as `args[0]`, so perhaps an ID was in there:

```
/*
 * sata_pkt_txlate structure contains info about resources allocated
 * for the packet
 * Address of this structure is stored in scsi_pkt.pkt_ha_private and
 * in sata_pkt.sata_hba_private fields, so all three strucures are
 * cross-linked, with sata_pkt_txlate as a centerpiece.
 */

typedef struct sata_pkt_txlate {
        struct sata_hba_inst    *txlt_sata_hba_inst;
        struct scsi_pkt         *txlt_scsi_pkt;
        struct sata_pkt         *txlt_sata_pkt;
[...]
```

I didn't see an ID member, but I did have the next best thing: The comment says that `sata_pkt_txlate` is *for the packet*. This isn't some data type shared by others; it's unique for the packet. So, I could use the memory address of this data type as my unique ID. The memory address is unique: Two variables can't store data on the same memory address. It's not as good as a unique ID, since the packet could be relocated in memory during the I/O, changing the ID. But that's unlikely, for a couple of reasons:

> Moving packet data around memory is a performance-expensive operation (memory I/O). The kernel stack is optimized for the opposite—“zero copy”—trying to avoid moving data around memory if at all possible.
>
> If an I/O is in-flight and the kernel changes the packet memory location, it would need to change all the other variables that refer to it, including interrupt callbacks. This could be difficult to do and would be avoided if possible.

So, I decided to try using the `spx` variables themselves as the unique ID to start with:

```
static int sata_hba_start(sata_pkt_txlate_t *spx, int *rval) ;
static void sata_pkt_free(sata_pkt_txlate_t *spx) ;
```

Available either as `args[0]` or even `arg0`—the `uint64_t` version—since I wasn’t referencing it to walk its members, but rather to refer to the memory address itself. If I hit a problem, I could try some of the other packet unique memory addresses in `sata_pkt_txlate`, such as `txlt_scsi_pkt` and `txlt_sata_pkt`.

I was now ready to try writing a basic DTrace script for measuring SATA command latency, `latency.d`:

```
1  #!/usr/sbin/dtrace -s
2
3  fbt::sata_hba_start:entry
4  {
5          start_time[arg0] = timestamp;
6  }
7
8  fbt::sata_pkt_free:entry
9  /start_time[arg0]/
10  {
11          this->delta = (timestamp - start_time[arg0]) / 1000;
12          @["Average SATA time (us):"] = avg(this->delta);
13          start_time[arg0] = 0;
14  }
```

Script sata_latency.d

Line 9 ensured that this packet was seen on `sata_hba_start()` and had its time stamp recorded (checking whether that key has a nonzero value). This also neatly solved the earlier concern with `sata_pkt_free()`. The possibility of it tracing invalid packets that were never sent to the HBA.

Testing

DTrace makes it easy to create statistics that appear useful, but under closer scrutiny creates more questions than answers, for example, measuring too few or too many I/Os for different workloads. It's crucial to double-check everything.

Given `latency.d`, I now performed sanity tests to see whether this latency looked correct. Here I performed random read I/O to a SATA disk and first checked latency with the system tool `iostat(1M)`:

```
# iostat -xnz 5
[...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
   81.0    0.0  648.0    0.0  0.0  1.0    0.0   12.3   0  99 c3t1d0
```

I was getting average service times of 12.3 ms.

Running the `latency.d` script, here's the output:

```
# latency.d
dtrace: script '/chapters/disk/sata/latency.d' matched 2 probes
^C

  Average SATA time (us):                                        12342
```

This also showed 12.3 ms—a perfect match (for this workload, anyway).

While I was getting the same information from `iostat(1M)` at the moment, I could customize `latency.d` in any direction desirable: showing latency as distribution plots, showing it for some types of SATA or SCSI commands only, and so on.

I repeated such testing for all the probes and arguments I used and brainstormed which different workload types might be processed differently by the driver to ensure that my driver-dependent, fbt-based script still handles them. (I spend much more time testing DTrace scripts than I ever do writing them. As the saying goes, "If it isn't tested, it doesn't work.")

Read the Source

At this point, I browsed through the source code to see whether I'd missed anything and to double-check that the functions I was tracing made sense. I checked block comments, function names, any calls for system logging or debug logging (since they are often descriptive and placed at logical points in the code), and anything else that looked interesting. I might have spotted procedures custom to this driver that I wouldn't normally think of tracing. I might also have seen special case functions for a particular type of I/O that I hadn't tested yet.

Browsing the source, I realized I'd made an incorrect assumption. I'd assumed that sata_hba_start() would trace all SATA commands issued to the SATA HBA. sata_hba_start() sends SATA commands using the following:

```
stat = (*SATA_START_FUNC(sata_hba_inst))(SATA_DIP(sata_hba_inst),
            spx->txlt_sata_pkt);
```

SATA_START_FUNC is a macro to ask the driver from sata_hba_inst to call its transport function, which is stored in the sata_tran_start member:

```
#define SATA_START_FUNC(sata_hba_inst) \
        sata_hba_inst->satahba_tran->sata_tran_start
```

The question was, do any other functions in SATA call SATA_START_FUNC and bypass sata_hba_start()? If so, I couldn't trace all SATA commands with sata_hba_start() alone.

By searching for SATA_START_FUNC in sata.c, I discovered that there were several other functions that called it, such as sata_set_dma_mode() and sata_set_cache_mode(). These events are uncommon, which is why I hadn't spotted the discrepancy in testing.

I began by adding the several other functions to my latency script so that it matched the beginning of *all* SATA commands. It started to become complex, so I searched for and found an alternative: the sata_tran_start member, which points to the specific SATA HBA start function. Although it is relative to the SATA HBA driver (nv_sata, ahci, and so on), there is only one such function for each, and it has the same arguments. Seaching the kernel code showed that they were as follows:

```
stran.sata_tran_start = nv_sata_start;
        sata_hba_tran->sata_tran_start = ahci_tran_start;
[...]
```

So, instead of tracing the several sata functions that may call sata_tran_start, I could trace the specific sata_tran_start functions themselves, such as nv_sata_start() and ahci_tran_start(). This made the DTrace script easier (fewer and simpler probes) but made it dependent on the SATA HBAs that it specifies. See the satalatency.d script for the final solution I wrote for this, which included the following:

```
/*
 * Trace SATA command start by probing the entry to the SATA HBA driver.  Four
 * different drivers are covered here; add yours here if it is missing.
 */
fbt::nv_sata_start:entry,
fbt::bcm_sata_start:entry,
fbt::ahci_tran_start:entry,
fbt::mv_start:entry
{
        start[arg1] = timestamp;
}
```

Conclusion

With any luck, your DTracing experiences will end with the second step: the discovery of a stable provider for the target of interest. But if one does not exist, there is much more that can be done, as shown by this case study.

Summary

This chapter showed many ways to observe useful disk I/O details using DTrace, at different layers of the I/O subsystem. Many of these scripts print statistics that were previously difficult or impossible to see without installing debug drivers or using tunables. With DTrace, you have the power to observe the entire disk I/O subsystem instantly, on demand, in *production*, and without installing custom drivers or rebooting.

The most important lesson from this chapter is that DTrace *can* observe all of these I/O subsystem components, in as much detail as needed. The last 80 pages showed numerous examples of this, not just as demonstrations of useful tools but also to illustrate the scope of DTrace. Our main objective here is simply to show you that it is possible to observe these details—even if you forget the specifics of how (you can refer to this book later) and even if all the fbt-based scripts stop working (you can try fixing them or find updates on the Web[23]).

23. And we may add stable providers for SCSI, SATA, and so on, so that these scripts can be written once and will always work.

5

File Systems

File systems—an integral part of any operating system—have long been one of the most difficult components to observe when analyzing performance. This is largely because of the way file system data and metadata caching are implemented in the kernel but also because, until now, we simply haven't had tools that can look into these kernel subsystems. Instead, we've analyzed slow I/O at the disk storage layer with tools such as iostat(1), even though this is many layers away from application latency. DTrace can be used to observe exactly how the file system responds to applications, how effective file system tuning is, and the internal operation of file system components. You can use it to answer questions such as the following.

- What files are being accessed, and how? By what or whom? Bytes, I/O counts?
- What is the source of file system latency? Is it disks, the code path, locks?
- How effective is prefetch/read-ahead? Should this be tuned?

As an example, rwsnoop is a DTrace-based tool, shipping with Mac OS X and OpenSolaris, that you can use to trace read and write system calls, along with the filename for file system I/O. The following shows sshd (the SSH daemon) accepting a login on Solaris:

```
# rwsnoop
  UID    PID CMD          D   BYTES FILE
    0 942611 sshd         R      70 <unknown>
    0 942611 sshd         R       0 <unknown>
```

continues

```
     0 942611 sshd          R    1444 /etc/gss/mech
     0 942611 sshd          R       0 /etc/gss/mech
     0 942611 sshd          R       0 /etc/krb5/krb5.conf
     0 942611 sshd          R    1894 /etc/crypto/pkcs11.conf
     0 942611 sshd          R       0 /etc/crypto/pkcs11.conf
     0 942611 sshd          R     336 /proc/942611/psinfo
     0 942611 sshd          R     553 /etc/nsswitch.conf
     0 942611 sshd          R       0 /etc/nsswitch.conf
     0 942611 sshd          R     916 /var/ak/etc/passwd
     0 942611 sshd          R       4 /.sunw/pkcs11_softtoken/objstore_info
     0 942611 sshd          R      16 /.sunw/pkcs11_softtoken/objstore_info
     0 942611 sshd          W      12 /devices/pseudo/random@0:urandom
     0 942611 sshd          R       0 /etc/krb5/krb5.conf
     0 942611 sshd          W      12 /devices/pseudo/random@0:urandom
     0 942611 sshd          R       0 /etc/krb5/krb5.conf
     0 942611 sshd          W      12 /devices/pseudo/random@0:urandom
     0 942611 sshd          R       0 /etc/krb5/krb5.conf
     0 942611 sshd          W      12 /devices/pseudo/random@0:urandom
     0 942611 sshd          W     520 <unknown>
[...]
```

Unlike `iosnoop` from Chapter 4, Disk I/O, the reads and writes shown previously may be served entirely from the file system in-memory cache, with no need for any corresponding physical disk I/O.

Since `rwsnoop` traces syscalls, it also catches reads and writes to non–file system targets, such as sockets for network I/O (the `<unknown>` filenames). Or DTrace can be used to drill down into the file system and catch only file system I/O, as shown in the "Scripts" section.

Capabilities

The file system functional diagram shown in Figure 5-1 represents the flow from user applications, through the major kernel subsystems, down to the storage subsystem. The path of a data or metadata disk operation may fall into any of the following:

1. Raw I/O (/dev/rdsk)
2. File system I/O
3. File system ops (mount/umount)
4. File system direct I/O (cache bypass)
5. File system I/O
6. Cache hits (reads)/writeback (writes)
7. Cache misses (reads)/writethrough (writes)
8. Physical disk I/O

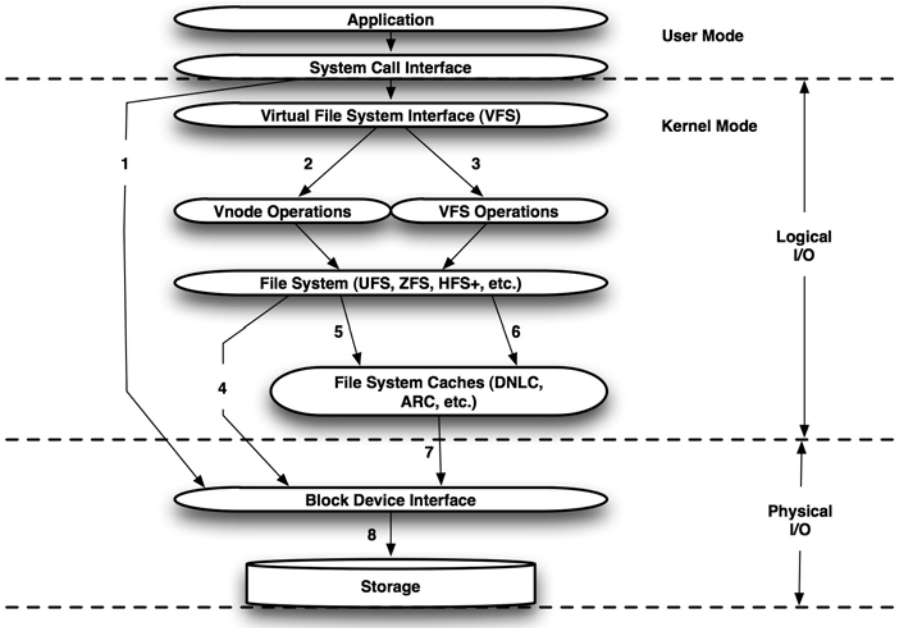


Figure 5-1 File system functional diagram

Figure 5-2 shows the logical flow of a file system read request processing through to completion. At each of the numbered items, we can use DTrace to answer questions, such as the following.

1. What are the requests? Type? Count? Read size? File offset?
2. What errors occurred? Why? For who/what?
3. How many reads were from prefetch/read ahead? (ZFS location shown.)
4. What was the cache hit rate? Per file system?
5. What is the latency of read, cache hit (request processing)?
6. What is the full request processing time (cache lookup + storage lookup)?
7. What is the volume of disk I/O? (How does it compare to 1?)
8. What is the disk I/O latency?
9. Did any disk errors occur?
10. Latency of I/O, cache miss?
11. Error latency? (May include disk retries.)

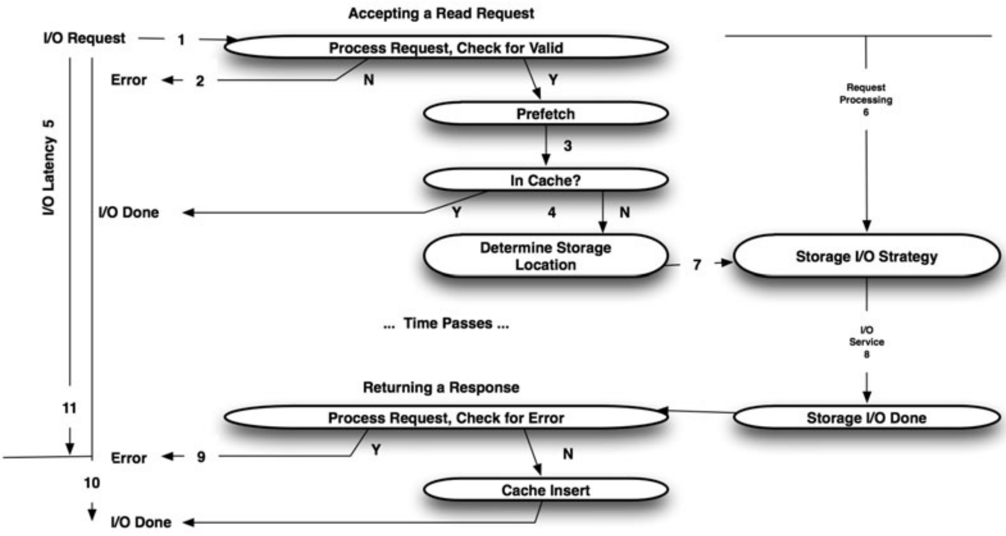


Figure 5-2 File system read operation

Figure 5-3 shows the logical flow of a file system write request processing through to completion. At each of the numbered items, we can use DTrace to answer questions, such as the following.

1. What are the requests? Type? Count? Write size? File offset?
2. What errors occurred? Why? For who/what?
3. How much of the write I/O was synchronous?
4. What is the latency of write, writeback (request processing)?
5. What is the full request processing time (cache insertion + storage lookup)?
6. What is the volume of disk I/O? (How does it compare to 1?)
7. What is the disk I/O latency for normal writes?
8. What is the disk I/O latency for synchronous writes (includes disk cache sync)?
9. Did any disk errors occur?
10. What is the latency of an I/O on a cache miss?
11. What is the error latency? (This may include disk retries.)

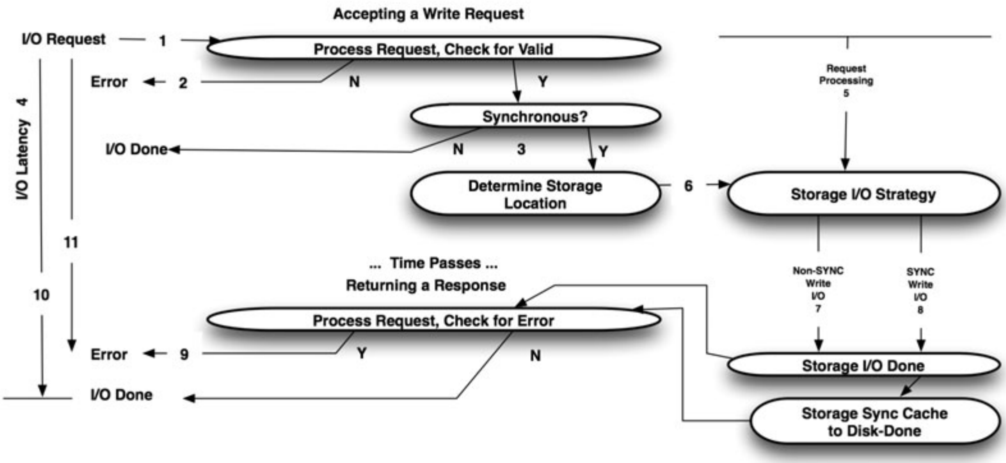


Figure 5-3 File system write operation

Logical vs. Physical I/O

Figure 5-1 labels I/O at the system call layer as "logical" and I/O at the disk layer as "physical." Logical I/O describes all requests to the file system, including those that return immediately from memory. Physical I/O consists of requests by the file system to its storage devices.

There are many reasons why the rate and volume of logical I/O may not match physical I/O, some of which may already be obvious from Figure 5-1. These include caching, read-ahead/prefetch, file system record size inflation, device sector size fragmentation, write cancellation, and asynchronous I/O. Each of these are described in the "Scripts" section for the `readtype.d` and `writetype.d` scripts, which trace and compare logical to physical I/O.

Strategy

The following approach will help you get started with disk I/O analysis using DTrace. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.

1. In addition to those DTrace tools, familiarize yourself with **existing file system statistical tools**. For example, on Solaris you can use `df(1M)` to list file system usage, as well as a new tool called `fsstat(1)` to show file system I/O types. You can use the metrics from these as starting points for customization with DTrace.

2. Locate or write tools to **generate known file system I/O**, such as running the dd command to create files with known numbers of write I/O and to read them back. Filebench can be used to generate sophisticated I/O. It is extremely helpful to have known workloads to check against.
3. **Customize** and write your own one-liners and scripts using the syscall provider. Then try the vminfo and sysinfo providers, if available.
4. Try the currently unstable **fsinfo provider** for more detailed file system scripts, and customize the fsinfo scripts in this chapter.
5. To dig deeper than these providers allow, familiarize yourself with how the kernel and user-land processes call file system I/O by examining stack backtraces (see the "One-Liners" section). Also refer to functional diagrams of the file system subsystem, such as the generic one shown earlier, and others for specific file system types. Check published kernel texts such as *Solaris Internals* (McDougall and Mauro, 2006) and *Mac OS X Internals* (Singh, 2006).
6. Examine **kernel internals** for file systems by using the fbt provider and referring to kernel source code (if available).

Checklist

Table 5-1 describes some potential problem areas with file systems, with suggestions on how you can use DTrace to troubleshoot them.

Table 5-1 File System I/O Checklist

Issue	Description
Volume	Applications may be performing a high volume of file system I/O, which could be avoided or optimized by changing their behavior, for example, by tuning I/O sizes and file locations (tmpfs instead of nfs, for example). The file system may break up I/O into multiple physical I/O of smaller sizes, inflating the IOPS. DTrace can be used to examine file system I/O by process, filename, I/O size, and application stack trace, to identify what files are being used, how, and why.
Latency	A variety of latencies should be examined when analyzing file system I/O: • Disk I/O wait, for reads and synchronous writes • Locking in the file system • Latency of the `open()` syscall • Large file deletion time Each of these can be examined using DTrace.

Table 5-1 File System I/O Checklist (*Continued*)

Issue	Description
Queueing	Use DTrace to examine the size and wait time for file system queues, such as queueing writes for later flushing to disk. Some file systems such as ZFS use a pipeline for all I/O, with certain stages serviced by multiple threads. High latency can occur if a pipeline stage becomes a bottleneck, for example, if compression is performed; this can be analyzed using DTrace.
Caches	File system performance can depend on cache performance: File systems may use multiple caches for different data types (directory names, inodes, metadata, data) and different algorithms for cache replacement and size. DTrace can be used to examine not just the hit and miss rate of caches, but what types of data are experiencing misses, what contents are being evicted, and other internals of cache behavior.
Errors	The file system interface can return errors in many situations: invalid file offsets, permission denied, file not found, and so on. Applications are supposed to catch and deal with these errors with them appropriately, but sometimes they silently fail. Errors returned by file systems can be identified and summarized using DTrace.
Configuration	File access can be tuned by flags, such as those on the `open()` syscall. DTrace can be used to check that the optimum flags are being used by the application, or if it needs to be configured differently.

Providers

Table 5-2 shows providers you can use to trace file system I/O.

Table 5-2 Providers for File System I/O

Provider	Description
syscall	Many syscalls operate on file systems (`open()`, `stat()`, `creat()`, and so on); some operate on file descriptors to file systems (`read()`, `write()`, and so on). By examining file system activity at the syscall interface, user-land context can be examined to see why the file system is being used, such as examining user stack backtraces.
vminfo	Virtual memory info provider. This includes file system page-in and page-out probes (file system disk I/O); however, these only provide number of pages and byte counts.
fsinfo	File system info provider: This is a representation of the VFS layer for the operating system and allows tracing of file system events across different file system types, with file information for each event. This isn't considered a stable provider as the VFS interface can change and is different for different OSs. However, it is unlikely to change rapidly.

continues

Table 5-2 Providers for File System I/O (*Continued*)

Provider	Description
vfs	Virtual File System provider: This is on FreeBSD only and shows VFS and name-cache operations.
io	Trace disk I/O event details including disk, bytes, and latency. Examining stack backtraces from `io:::start` shows why file systems are calling disk I/O.
fbt	Function Boundary Tracing provider. This allows file system internals to be examined in detail, including the operation of file system caches and read ahead. This has an unstable interface and will change between releases of the operating system and file systems, meaning that scripts based on fbt may need to be slightly rewritten for each such update.

Check your operating system to see which providers are available; at the very least, syscall and fbt should be available, which provide a level of coverage of everything.

The vminfo and io providers should also be available on all versions of Solaris 10 and Mac OS X. fsinfo was added to Solaris 10 6/06 (update 2) and Solaris Nevada build 38 and is not yet available on Mac OS X.

fsinfo Provider

The fsinfo provider traces logical file system access. It exports the VFS vnode interface, a private interface for kernel file systems, so fsinfo is considered an unstable provider.

Because the vnode operations it traces are descriptive and resemble many well-known syscalls (`open()`, `close()`, `read()`, `write()`, and so on), this interface provides a generic view of what different file systems are doing and has been exported as the DTrace fsinfo provider.

Listing the fsinfo provider probes on a recent version of Solaris Nevada, we get the following results:

```
# dtrace -ln fsinfo:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
30648     fsinfo           genunix                       fop_vnevent vnevent
30649     fsinfo           genunix                       fop_shrlock shrlock
30650     fsinfo           genunix                    fop_getsecattr getsecattr
30651     fsinfo           genunix                    fop_setsecattr setsecattr
30652     fsinfo           genunix                       fop_dispose dispose
30653     fsinfo           genunix                       fop_dumpctl dumpctl
30654     fsinfo           genunix                        fop_pageio pageio
30655     fsinfo           genunix                      fop_pathconf pathconf
30656     fsinfo           genunix                          fop_dump dump
30657     fsinfo           genunix                          fop_poll poll
```

```
  30658    fsinfo          genunix                           fop_delmap delmap
  30659    fsinfo          genunix                           fop_addmap addmap
  30660    fsinfo          genunix                              fop_map map
  30661    fsinfo          genunix                          fop_putpage putpage
  30662    fsinfo          genunix                          fop_getpage getpage
  30663    fsinfo          genunix                           fop_realvp realvp
  30664    fsinfo          genunix                            fop_space space
  30665    fsinfo          genunix                           fop_frlock frlock
  30666    fsinfo          genunix                              fop_cmp cmp
  30667    fsinfo          genunix                             fop_seek seek
  30668    fsinfo          genunix                         fop_rwunlock rwunlock
  30669    fsinfo          genunix                           fop_rwlock rwlock
  30670    fsinfo          genunix                              fop_fid fid
  30671    fsinfo          genunix                         fop_inactive inactive
  30672    fsinfo          genunix                            fop_fsync fsync
  30673    fsinfo          genunix                         fop_readlink readlink
  30674    fsinfo          genunix                          fop_symlink symlink
  30675    fsinfo          genunix                          fop_readdir readdir
  30676    fsinfo          genunix                            fop_rmdir rmdir
  30677    fsinfo          genunix                            fop_mkdir mkdir
  30678    fsinfo          genunix                           fop_rename rename
  30679    fsinfo          genunix                             fop_link link
  30680    fsinfo          genunix                           fop_remove remove
  30681    fsinfo          genunix                           fop_create create
  30682    fsinfo          genunix                           fop_lookup lookup
  30683    fsinfo          genunix                           fop_access access
  30684    fsinfo          genunix                          fop_setattr setattr
  30685    fsinfo          genunix                          fop_getattr getattr
  30686    fsinfo          genunix                            fop_setfl setfl
  30687    fsinfo          genunix                            fop_ioctl ioctl
  30688    fsinfo          genunix                            fop_write write
  30689    fsinfo          genunix                             fop_read read
  30690    fsinfo          genunix                            fop_close close
  30691    fsinfo          genunix                             fop_open open
```

Table 5-3 fsinfo Probes

Probe	Description
`open`	Attempts to open the file described in the `args[0] fileinfo_t`
`close`	Closes the file described in the `args[0] fileinfo_t`
`read`	Attempts to read `arg1` bytes from the file in `args[0] fileinfo_t`
`write`	Attempts to write `arg1` bytes to the file in `args[0] fileinfo_t`
`fsync`	Calls fsync to synronize the file in `args[0] fileinfo_t`

A selection of these probes is described in Table 5-3.

fileinfo_t

The `fileinfo` structure contains members to describe the file, file system, and open flags of the file that the fsinfo operation is performed on. Some of these members may not be available for particular probes and return `<unknown>`, `<none>`, or 0:

```
typedef struct fileinfo {
        string fi_name;                 /* name (basename of fi_pathname) */
        string fi_dirname;              /* directory (dirname of fi_pathname) */
        string fi_pathname;             /* full pathname */
        offset_t fi_offset;             /* offset within file */
        string fi_fs;                   /* file system */
        string fi_mount;                /* mount point of file system */
        int fi_oflags;                  /* open(2) flags for file descriptor */
} fileinfo_t;
```

These are translated from the kernel vnode. The `fileinfo_t` structure is also available as the file descriptor array, `fds[]`, which provides convenient file information by file descriptor number. See the one-liners for examples of its usage.

io Provider

The io provider traces physical I/O and was described in Chapter 4.

One-Liners

These one-liners are organized by provider.

syscall Provider

Some of these use the `fds[]` array, which was a later addition to DTrace; for an example of similar functionality predating `fds[]`, see the `rwsnoop` script.

For the one-liners tracing `read(2)` and `write(2)` system calls, be aware that variants may exist (`readv()`, `pread()`, `pread64()`); use the "Count read/write syscalls by syscall type" one-liner to identify which are being used. Also note that these match all reads and writes, whether they are file system based or not, unless matched in a predicate (see the "zfs" one-liner).

Trace file opens with process name:

```
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Trace file `creat()` calls with file and process name:

```
dtrace -n 'syscall::creat*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Frequency count `stat()` file calls:

```
dtrace -n 'syscall::stat*:entry { @[copyinstr(arg0)] = count(); }'
```

Tracing the `cd(1) command`:

```
dtrace -n 'syscall::chdir:entry { printf("%s -> %s", cwd, copyinstr(arg0)); }'
```

Count read/write syscalls by syscall type:

```
dtrace -n 'syscall::*read*:entry,syscall::*write*:entry { @[probefunc] = count(); }'
```

Syscall `read(2)` by filename:

```
dtrace -n 'syscall::read:entry { @[fds[arg0].fi_pathname] = count(); }'
```

Syscall `write(2)` by filename:

```
dtrace -n 'syscall::write:entry { @[fds[arg0].fi_pathname] = count(); }'
```

Syscall `read(2)` by file system type:

```
dtrace -n 'syscall::read:entry { @[fds[arg0].fi_fs] = count(); }'
```

Syscall `write(2)` by file system type:

```
dtrace -n 'syscall::write:entry { @[fds[arg0].fi_fs] = count(); }'
```

Syscall `read(2)` by process name for the zfs file system only:

```
dtrace -n 'syscall::read:entry /fds[arg0].fi_fs == "zfs"/ { @[execname] = count(); }'
```

Syscall `write(2)` by process name and file system type:

```
dtrace -n 'syscall::write:entry { @[execname, fds[arg0].fi_fs] = count(); }
    END { printa("%18s %16s %16@d\n", @); }'
```

vminfo Provider

This processes paging in from the file system:

```
dtrace -n 'vminfo:::fspgin { @[execname] = sum(arg0); }'
```

fsinfo Provider

You can count file system calls by VFS operation:

```
dtrace -n 'fsinfo::: { @[probename] = count(); }'
```

You can count file system calls by mountpoint:

```
dtrace -n 'fsinfo::: { @[args[0]->fi_mount] = count(); }'
```

Bytes read by filename:

```
dtrace -n 'fsinfo:::read { @[args[0]->fi_pathname] = sum(arg1); }'
```

Bytes written by filename:

```
dtrace -n 'fsinfo:::write { @[args[0]->fi_pathname] = sum(arg1); }'
```

Read I/O size distribution by file system mountpoint:

```
dtrace -n 'fsinfo:::read { @[args[0]->fi_mount] = quantize(arg1); }'
```

Write I/O size distribution by file system mountpoint:

```
dtrace -n 'fsinfo:::write { @[args[0]->fi_mount] = quantize(arg1); }'
```

vfs Provider

Count file system calls by VFS operation:

```
dtrace -n 'vfs:vop::entry { @[probefunc] = count(); }'
```

Namecache hit/miss statistics:

```
dtrace -n 'vfs:namecache:lookup: { @[probename] = count(); }'
```

sdt Provider

You can find out who is reading from the ZFS ARC (in-DRAM cache):

```
dtrace -n 'sdt:::arc-hit,sdt:::arc-miss { @[stack()] = count(); }'
```

fbt Provider

The fbt provider instruments a particular operating system and version; these one-liners may therefore require modifications to match the software version you are running.

VFS: You can count file system calls at the fop interface (Solaris):

```
dtrace -n 'fbt::fop_*:entry { @[probefunc] = count(); }'
```

VFS: You can count file system calls at the VNOP interface (Mac OS X):

```
dtrace -n 'fbt::VNOP_*:entry { @[probefunc] = count(); }'
```

VFS: You can count file system calls at the VOP interface (FreeBSD):

```
dtrace -n 'fbt::VOP_*:entry { @[probefunc] = count(); }'
```

ZFS: You can show SPA sync with pool name and TXG number:

```
dtrace -n 'fbt:zfs:spa_sync:entry
{ printf("%s %d", stringof(args[0]->spa_name), arg1); }'
```

One-Liners: syscall Provider Examples

Trace File Opens with Process Name

Tracing opens can be a quick way of getting to know software. Software will often call `open()` on config files, log files, and device files. Sometimes tracing `open()` is a quicker way to find where config and log files exist than to read through the product documentation.

```
# dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
 29  87276                        open:entry dmake /var/ld/ld.config
 29  87276                        open:entry dmake /lib/libnsl.so.1
 29  87276                        open:entry dmake /lib/libsocket.so.1
 29  87276                        open:entry dmake /lib/librt.so.1
 29  87276                        open:entry dmake /lib/libm.so.1
 29  87276                        open:entry dmake /lib/libc.so.1
 29  87672                      open64:entry dmake /var/run/name_service_door
 29  87276                        open:entry dmake /etc/nsswitch.conf
 12  87276                        open:entry sh /var/ld/ld.config
 12  87276                        open:entry sh /lib/libc.so.1
dtrace: error on enabled probe ID 1 (ID 87672: syscall::open64:entry): invalid address
 (0x8225aff) in action #2 at DIF offset 28
 12  87276                        open:entry sh /var/ld/ld.config
 12  87276                        open:entry sh /lib/libc.so.1
[...]
```

The probe definition uses `open*` so that both `open()` and `open64()` versions are traced. This one-liner has caught a software build in progress; the process names `dmake` and `sh` can be seen, and the files they were opening are mostly library files under `/lib`.

The `dtrace` error is likely due to `copyinstr()` operating on a text string that hasn't been faulted into the process's virtual memory address space yet. The page fault would happen during the `open()` syscall, but we've traced it before it has happened. This can be solved by saving the address on `open*:entry` and using `copyinstr()` on `open*:return`, after the string is in memory.

Trace File creat() Calls with Process Name

This also caught a software build in progress. Here the `cp` command is creating files as part of the build. The Bourne shell `sh` also appears to be creating `/dev/null`; this is happening as part of shell redirection.

```
# dtrace -n 'syscall::creat*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
dtrace: description 'syscall::creat*:entry ' matched 2 probes
```

```
CPU     ID                    FUNCTION:NAME
 25  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/misc/amd64/xpv_autoconfig
 31  87670                    creat64:entry sh /dev/null
  0  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/drv/xdf
 20  87670                    creat64:entry sh /dev/null
 26  87670                    creat64:entry sh /dev/null
 27  87670                    creat64:entry sh /dev/null
 31  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/usr/lib/llib-l300.ln
  0  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/kernel/drv/amd64/iwscn
 12  87670                    creat64:entry cp /builds/brendan/ak-on-new/proto/root_i3
86/platform/i86xpv/kernel/drv/xnf
 16  87670                    creat64:entry sh obj32/ao_mca_disp.c
[...]
```

Frequency Count stat() Files

As a demonstration of frequency counting instead of tracing and of examining the `stat()` syscall, this frequency counts filenames from `stat()`:

```
# dtrace -n 'syscall::stat*:entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'syscall::stat*:entry ' matched 5 probes
^C

  /builds/brendan/ak-on-new/proto/root_i386/kernel/drv/amd64/mxfe/mxfe
1
  /builds/brendan/ak-on-new/proto/root_i386/kernel/drv/amd64/rtls/rtls
1
  /builds/brendan/ak-on-new/proto/root_i386/usr/kernel/drv/ii/ii                1
  /lib/libmakestate.so.1                                            1
  /tmp/dmake.stdout.10533.189.ejaOKu                                1
[...output truncated...]
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/libmd5.so.1               105
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/sys/libc.so.1              105
  /ws/onnv-tools/SUNWspro/SS12/prod/lib/sys/libmd5.so.1              105
  /ws/onnv-tools/SUNWspro/SS12/prod/bin/../lib/libc.so.1               106
  /ws/onnv-tools/SUNWspro/SS12/prod/bin/../lib/lib_I_dbg_gen.so.1             107
  /lib/libm.so.1                                                  112
  /lib/libelf.so.1                                                136
  /lib/libdl.so.1                                                 151
  /lib/libc.so.1                                                  427
  /tmp                                                            638
```

During tracing, `stat()` was called on `/tmp` 638 times. A wildcard is used in the probe name so that this one-liner matches both `stat()` and `stat64()`; however, applications could be using other variants such as `xstat()` that this isn't matching.

Tracing cd

You can trace the current working directory (pwd) and chdir directory (cd) using the following one-liner:

```
# dtrace -n 'syscall::chdir:entry { printf("%s -> %s", cwd, copyinstr(arg0)); }'
dtrace: description 'syscall::chdir:entry ' matched 1 probe
CPU     ID      FUNCTION:NAME
  4  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> aac
  5  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> amd64_gart
  8  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> amr
  9  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> agptarget
 12  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> aggr
 12  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> agpgart
 16  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> ahci
 16  87290       chdir:entry /builds/brendan/ak-on-new/usr/src/uts/intel -> arp
[...]
```

This output shows a software build iterating over subdirectories.

Reads by File System Type

During this build, tmpfs is currently receiving the most reads: 128,645 during this trace, followed by ZFS at 65,919.

```
# dtrace -n 'syscall::read:entry { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
^C

  specfs                                                             22
  sockfs                                                             28
  proc                                                              103
  <none>                                                            136
  nfs4                                                              304
  fifofs                                                           1571
  zfs                                                             65919
  tmpfs                                                          128645
```

Note that this one-liner is matching only the read variant of the read() syscall. On Solaris, applications may be calling readv(), pread(), or pread64(); Mac OS X has readv(), pread(), read_nocancel(), and pread_nocancel(); and FreeBSD has more, including aio_read(). You can match all of these using wildcards:

```
solaris# dtrace -ln 'syscall::*read*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
87272    syscall                                                read entry
87418    syscall                                            readlink entry
87472    syscall                                               readv entry
87574    syscall                                               pread entry
87666    syscall                                             pread64 entry
```

However, this also matches `readlink()`, and our earlier one-liner assumes that `arg0` is the file descriptor, which is not the case for `readlink()`. Tracing all read types properly will require a short script rather than a one-liner.

Writes by File System Type

This one-liner matches all variants of write, assuming that `arg0` is the file descriptor. In this example, most of the writes were to tmpfs (`/tmp`).

```
# dtrace -n 'syscall::*write*:entry { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

  specfs                                                            2
  nfs4                                                             47
  sockfs                                                           55
  zfs                                                             154
  fifofs                                                          243
  tmpfs                                                         22245
```

Writes by Process Name and File System Type

This example extends the previous one-liner to include the process name:

```
# dtrace -n 'syscall::write:entry { @[execname, fds[arg0].fi_fs] = count(); }
END { printa("%18s %16s %16@d\n", @); }'
dtrace: description 'syscall::write:entry ' matched 2 probes
^C
CPU     ID                    FUNCTION:NAME
 25      2                             :END               ar         zfs            1
            dtrace           specfs                1
                sh           fifofs                1
              sshd           specfs                1
  ssh-socks5-proxy           fifofs                2
             uname           fifofs                3
               sed              zfs                4
               ssh           fifofs               10
             strip              zfs               15
[...truncated...]
               gas            tmpfs              830
             acomp            tmpfs             2072
               ube            tmpfs             2487
             ir2hf            tmpfs             2608
             iropt            tmpfs             3364
```

Now we can see the processes that were writing to tmpfs: `iropt`, `ir2hf`, and so on.

One-Liners: vminfo Provider Examples

Processes Paging in from the File System

The vminfo provider has a probe for file system page-ins, which can give a very rough idea of which processes are reading from disk via a file system:

```
# dtrace -n 'vminfo:::fspgin { @[execname] = sum(arg0); }'
dtrace: description 'vminfo:::fspgin ' matched 1 probe
^C

  dmake                                                             1
  scp                                                               2
  sched                                                            42
```

This worked a little: Both `dmake` and `scp` are responsible for paging in file system data. However, it has identified `sched` (the kernel) as responsible for the most page-ins. This could be because of read-ahead occurring in kernel context; more DTrace will be required to explain where the `sched` page-ins were from.

One-Liners: fsinfo Provider Examples

File System Calls by fs Operation

This uses the fsinfo provider, if available. Since it traces file system activity at the VFS layer, it will see activity from all file system types: ZFS, UFS, HSFS, and so on.

```
# dtrace -n 'fsinfo::: { @[probename] = count(); }'
dtrace: description 'fsinfo::: ' matched 44 probes
^C

  rename                                                            2
  symlink                                                           4
  create                                                            6
  getsecattr                                                        6
  seek                                                              8
  remove                                                           10
  poll                                                             40
  readlink                                                         40
  write                                                            42
  realvp                                                           52
  map                                                             144
  read                                                            171
  addmap                                                          192
  open                                                            193
  delmap                                                          194
  close                                                           213
  readdir                                                         225
  dispose                                                         230
  access                                                          248
  ioctl                                                           421
  rwlock                                                          436
  rwunlock                                                        436
```

```
   getpage                                                      1700
   getattr                                                      3221
   cmp                                                         48342
   putpage                                                     77557
   inactive                                                    80786
   lookup                                                      86059
```

The most frequent vnode operation was `lookup()`, called 86,059 times while this one-liner was tracing.

File System Calls by Mountpoint

The fsinfo provider has `fileinfo_t` as `args[0]`. Here the mountpoint is frequency counted by fsinfo probe call, to get a rough idea of how busy (by call count) file systems are as follows:

```
# dtrace -n 'fsinfo::: { @[args[0]->fi_mount] = count(); }'
dtrace: description 'fsinfo::: ' matched 44 probes
^C

  /home                                                            8
  /builds/bmc                                                      9
  /var/run                                                        11
  /builds/ahl                                                     24
  /home/brendan                                                   24
  /etc/svc/volatile                                               47
  /etc/svc                                                        50
  /var                                                            94
  /net/fw/export/install                                         176
  /ws                                                            252
  /lib/libc.so.1                                                 272
  /etc/mnttab                                                    388
  /ws/onnv-tools                                                1759
  /builds/brendan                                              17017
  /tmp                                                        156487
  /                                                           580819
```

Even though I'm doing a source build in `/builds/brendan`, it's the root file system on / that has received the most file system calls.

Bytes Read by Filename

The fsinfo provider gives an abstracted file system view that isn't dependent on syscall variants such as `read()`, `pread()`, `pread64()`, and so on.

```
# dtrace -n 'fsinfo:::read { @[args[0]->fi_pathname] = sum(arg1); }'
dtrace: description 'fsinfo:::read ' matched 1 probe
^C

  /usr/bin/chmod                                                 317
  /home/brendan/.make.machines                                   572
```

continues

```
  /usr/bin/chown                                                   951
  <unknown>                                                       1176
  /usr/bin/chgrp                                                  1585
  /usr/bin/mv                                                     1585
[...output truncated...]
  /builds/brendan/ak-on-new/usr/src/uts/intel/Makefile.rules           325056
  /builds/brendan/ak-on-new/usr/src/uts/intel/Makefile.intel.shared           415752
  /builds/brendan/ak-on-new/usr/src/uts/intel/arn/.make.state           515044
  /builds/brendan/ak-on-new/usr/src/uts/Makefile.uts          538440
  /builds/brendan/ak-on-new/usr/src/Makefile.master           759744
  /builds/brendan/ak-on-new/usr/src/uts/intel/ata/.make.state           781904
  /builds/brendan/ak-on-new/usr/src/uts/common/Makefile.files           991896
  /builds/brendan/ak-on-new/usr/src/uts/common/Makefile.rules          1668528
  /builds/brendan/ak-on-new/usr/src/uts/intel/genunix/.make.state          5899453
```

The file being read the most is a `.make.state` file: During tracing, more than 5MB was read from the file. The fsinfo provider traces these reads to the file system: The file may have been entirely cached in DRAM or read from disk. To determine how the read was satisfied by the file system, we'll need to DTrace further down the I/O stack (see the "Scripts" section and Chapter 4, Disk I/O).

Bytes Written by Filename

During tracing, a `.make.state.tmp` file was written to the most, with more than 1MB of writes. As with reads, this is writing to the file system. This may not write to disk until sometime later, when the file system flushes dirty data.

```
# dtrace -n 'fsinfo:::write { @[args[0]->fi_pathname] = sum(arg1); }'
dtrace: description 'fsinfo:::write ' matched 1 probe
^C

  /tmp/DAA1RaGkd                                                    22
  /tmp/DAA5JaO6c                                                    22
[...truncated...]
  /tmp/iroptEAA.1524.dNaG.c                                     250588
  /tmp/acompBAA.1443.MGay0c                                     305541
  /tmp/iroptDAA.1443.OGay0c                                     331906
  /tmp/acompBAA.1524.aNaG.c                                     343015
  /tmp/iroptDAA.1524.cNaG.c                                     382413
  /builds/brendan/ak-on-new/usr/src/cmd/fs.d/.make.state.tmp          1318590
```

Read I/O Size Distribution by File System Mountpoint

This output shows a distribution plot of read size by file system. The `/builds/brendan` file system was usually read at between 1,024 and 131,072 bytes per read. The largest read was in the 1MB to 2MB range.

```
# dtrace -n 'fsinfo:::read { @[args[0]->fi_mount] = quantize(arg1); }'
dtrace: description 'fsinfo:::read ' matched 1 probe
^C
```

```
  /builds/bmc
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
               1 |                                         0

[...output truncated...]

  /builds/brendan
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@                                        15
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |@@                                       28
             128 |                                         0
             256 |                                         0
             512 |@@                                       28
            1024 |@@@@@@@                                  93
            2048 |@@@@                                     52
            4096 |@@@@@@                                   87
            8192 |@@@@@@@                                  94
           16384 |@@@@@@@@                                 109
           32768 |@@                                       31
           65536 |@@                                       30
          131072 |                                         0
          262144 |                                         2
          524288 |                                         1
         1048576 |                                         1
         2097152 |                                         0
```

Write I/O Size Distribution by File System Mountpoint

During tracing, `/tmp` was written to the most (listed last), mostly with I/O sizes between 4KB and 8KB.

```
# dtrace -n 'fsinfo:::write { @[args[0]->fi_mount] = quantize(arg1); }'
dtrace: description 'fsinfo:::write ' matched 1 probe
^C

  /etc/svc/volatile
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 34
             512 |                                         0
[...]

  /tmp
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |                                         1
               8 |                                         4
              16 |@@@@                                     121
              32 |@@@@                                     133
              64 |@@                                       56
             128 |@@                                       51
```

continues

```
           256 |@                                        46
           512 |@                                        39
          1024 |@                                        32
          2048 |@@                                       52
          4096 |@@@@@@@@@@@@@@@@@@@@@@@@                 820
          8192 |                                         0
```

One-Liners: sdt Provider Examples

Who Is Reading from the ZFS ARC?

This shows who is performing reads to the ZFS ARC (the in-DRAM file system cache for ZFS) by counting the stack backtraces for all ARC accesses. It uses SDT probes, which have been in the ZFS ARC code for a while:

```
# dtrace -n 'sdt:::arc-hit,sdt:::arc-miss { @[stack()] = count(); }'
dtrace: description 'sdt:::arc-hit,sdt:::arc-miss ' matched 3 probes
^C
[...]

              zfs`arc_read+0x75
              zfs`dbuf_prefetch+0x131
              zfs`dmu_prefetch+0x8f
              zfs`zfs_readdir+0x4a2
              genunix`fop_readdir+0xab
              genunix`getdents64+0xbc
              unix`sys_syscall32+0x101
              245

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold_array_by_dnode+0x195
              zfs`dmu_buf_hold_array+0x73
              zfs`dmu_read_uio+0x4d
              zfs`zfs_read+0x19a
              genunix`fop_read+0x6b
              genunix`read+0x2b8
              genunix`read32+0x22
              unix`sys_syscall32+0x101
              457

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold+0x75
              zfs`zap_lockdir+0x67
              zfs`zap_cursor_retrieve+0x74
              zfs`zfs_readdir+0x29e
              genunix`fop_readdir+0xab
              genunix`getdents64+0xbc
              unix`sys_syscall32+0x101
             1004

              zfs`dbuf_hold_impl+0xea
              zfs`dbuf_hold+0x2e
              zfs`dmu_buf_hold+0x75
              zfs`zap_lockdir+0x67
              zfs`zap_lookup_norm+0x55
              zfs`zap_lookup+0x2d
```

```
                zfs`zfs_match_find+0xfd
                zfs`zfs_dirent_lock+0x3d1
                zfs`zfs_dirlook+0xd9
                zfs`zfs_lookup+0x104
                genunix`fop_lookup+0xed
                genunix`lookuppnvp+0x3a3
                genunix`lookuppnat+0x12c
                genunix`lookupnameat+0x91
                genunix`cstatat_getvp+0x164
                genunix`cstatat64_32+0x82
                genunix`lstat64_32+0x31
                unix`sys_syscall32+0x101
               2907
```

This output is interesting because it demonstrates four different types of ZFS ARC read. Each stack is, in order, as follows.

1. `prefetch read`: ZFS performs prefetch before reading from the ARC. Some of the prefetch requests will actually just be cache hits; only the prefetch requests that miss the ARC will pull data from disk.
2. `syscall read`: Most likely a process reading from a file on ZFS.
3. `read dir`: Fetching directory contents.
4. `stat`: Fetching file information.

Scripts

Table 5-4 summarizes the scripts that follow and the providers they use.

Table 5-4 Script Summary

Script	Target	Description	Providers
`sysfs.d`	Syscalls	Shows reads and writes by process and mountpoint	syscall
`fsrwcount.d`	Syscalls	Counts read/write syscalls by file system and type	syscall
`fsrwtime.d`	Syscalls	Measures time in read/write syscalls by file system	syscall
`fsrtpk.d`	Syscalls	Measures file system read time per kilobyte	syscall
`rwsnoop`	Syscalls	Traces syscall read and writes, with FS details	syscall
`mmap.d`	Syscalls	Traces `mmap()` of files with details	syscall
`fserrors.d`	Syscalls	Shows file system syscall errors	syscall

continues

Table 5-4 Script Summary (*Continued*)

Script	Target	Description	Providers
fswho.d[1]	VFS	Summarizes processes and file read/writes	fsinfo
readtype.d[1]	VFS	Compares logical vs. physical file system reads	fsinfo, io
writetype.d[1]	VFS	Compares logical vs. physical file system writes	fsinfo, io
fssnoop.d	VFS	Traces file system calls using fsinfo	fsinfo
solvfssnoop.d	VFS	Traces file system calls using fbt on Solaris	fbt
macvfssnoop.d	VFS	Traces file system calls using fbt on Mac OS X	fbt
vfssnoop.d	VFS	Traces file system calls using vfs on FreeBSD	vfs
sollife.d	VFS	Shows file creation and deletion on Solaris	fbt
maclife.d	VFS	Shows file creation and deletion on Mac OS X	fbt
vfslife.d	VFS	Shows file creation and deletion on FreeBSD	vfs
dnlcps.d	VFS	Shows Directory Name Lookup Cache hits by process[2]	fbt
fsflush_cpu.d	VFS	Shows file system flush tracer CPU time[2]	fbt
fsflush.d	VFS	Shows file system flush statistics[2]	profile
ufssnoop.d	UFS	Traces UFS calls directly using fbt[2]	fbt
ufsreadahead.d	UFS	Shows UFS read-ahead rates for sequential I/O[2]	fbt
ufsimiss.d	UFS	Traces UFS inode cache misses with details[2]	fbt
zfssnoop.d	ZFS	Traces ZFS calls directly using fbt[2]	fbt
zfsslower.d	ZFS	Traces slow HFS+ read/writes[2]	fbt
zioprint.d	ZFS	Shows ZIO event dump[2]	fbt
ziosnoop.d	ZFS	Shows ZIO event tracing, detailed[2]	fbt
ziotype.d	ZFS	Shows ZIO type summary by pool[2]	fbt
perturbation.d	ZFS	Shows ZFS read/write time during given perturbation[2]	fbt
spasync.d	ZFS	Shows SPA sync tracing with details[2]	fbt
hfssnoop.d	HFS+	Traces HFS+ calls directly using fbt[3]	fbt
hfsslower.d	HFS+	Traces slow HFS+ read/writes[3]	fbt
hfsfileread.d	HFS+	Shows logical/physical reads by file[3]	fbt
pcfsrw.d	PCFS	Traces pcfs (FAT16/32) read/writes[2]	fbt
cdrom.d	HSFS	Traces CDROM insertion and mount[2]	fbt
dvd.d	UDFS	Traces DVD insertion and mount[2]	fbt
nfswizard.d	NFS	Summarizes NFS performance client-side[2]	io

Table 5-4 Script Summary (*Continued*)

Script	Target	Description	Providers
`nfs3sizes.d`	NFSv3	Shows NFSv3 logical vs physical read sizes[2]	fbt
`nfs3fileread.d`	NFSv3	Shows NFSv3 logical vs physical reads by file[2]	fbt
`tmpusers.d`	TMPFS	Shows users of `/tmp` and `tmpfs` by tracing `open()`[2]	fbt
`tmpgetpage.d`	TMPFS	Measures whether tmpfs paging is occurring, with I/O time[2]	fbt

[1] This uses the fsinfo provider, currently available only on Oracle Solaris.

[2] This is written for Oracle Solaris.

[3] This is written for Apple Mac OS X.

There is an emphasis on the syscall and VFS layer scripts, since these can be used on any underlying file system type.

Note that the fbt provider is considered an "unstable" interface, because it instruments a specific operating system or application version. For this reason, scripts that use the fbt provider may require changes to match the version of the software you are using. These scripts have been included here as examples of D programming and of the kind of data that DTrace can provide for each of these topics. See Chapter 12, Kernel, for more discussion about using the fbt provider.

Syscall Provider

File system tracing scripts based on the syscall provider are generic and work across all file systems. At the syscall level, you can see "logical" file system I/O, the I/O that the application requests from the file system. Actual disk I/O occurs after file system processing and may not match the requested logical I/O (for example, rounding I/O size up to the file system block size).

sysfs.d

The `sysfs.d` script traces read and write syscalls to show which process is performing reads and writes on which file system.

Script

This script is written to work on both Solaris and Mac OS X. Matching all the possible `read()` variants (`read()`, `readv()`, `pread()`, `pread64()`, `read_nocancel()`, and so on) for Solaris and Mac OS X proved a little tricky and led to the probe definitions on lines 11 to 14. Attempting to match `syscall::*read*:entry` doesn't work, because it matches `readlink()` and pthread syscalls (on Mac OS X), neither of which we are trying to trace (we want a `read()` style syscall with a file descriptor as `arg0`, for line 17 to use).

The `-Z` option prevents DTrace on Solaris complaining about line 14, which is just there for the Mac OS X `read_nocancel()` variants. Without it, this script wouldn't execute because DTrace would fail to find probes for `syscall::*read*nocancel:entry`.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           printf("Tracing... Hit Ctrl-C to end.\n");
8   }
9
10  /* trace read() variants, but not readlink() or __pthread*() (macosx) */
11  syscall::read:entry,
12  syscall::readv:entry,
13  syscall::pread*:entry,
14  syscall::*read*nocancel:entry,
15  syscall::*write*:entry
16  {
17          @[execname, probefunc, fds[arg0].fi_mount] = count();
18  }
19
20  dtrace:::END
21  {
22          printf("  %-16s %-16s %-30s %7s\n", "PROCESS", "SYSCALL",
23              "MOUNTPOINT", "COUNT");
24          printa("  %-16s %-16s %-30s %@7d\n", @);
25  }
```

Script sysfs.d

Example

This was executed on a software build server. The busiest process name during tracing was `diff`, performing reads on the `/ws/ak-on-gate/public` file system. This was probably multiple `diff(1)` commands; the `sysfs.d` script could be modified to include a PID if it was desirable to split up the PIDs (although in this case it helps to aggregate the build processes together).

Some of the reads and writes to the `/` mountpoint may have been to device paths in `/dev`, including `/dev/tty` (terminal); to differentiate between these and I/O to the root file system, enhance the script to include a column for `fds[arg0].fi_fs`—the file system type (see `fsrwcount.d`).

```
# sysfs.d
Tracing... Hit Ctrl-C to end.
^C
```

```
  PROCESS          SYSCALL          MOUNTPOINT                          COUNT
  hg               write            /devices                                1
  in.mpathd        read             /                                       1
  in.mpathd        write            /                                       1
[...truncated...]
  nawk             write            /tmp                                   36
  dmake            write            /builds/brendan                        40
  nawk             write            /ws/ak-on-gate/public                  50
  dmake            read             /var                                   54
  codereview       write            /tmp                                   61
  ksh93            write            /ws/ak-on-gate/public                  65
  expand           read             /                                      69
  nawk             read             /                                      69
  expand           write            /                                      72
  sed              read             /tmp                                  100
  nawk             read             /tmp                                  113
  dmake            read             /                                     209
  dmake            read             /builds/brendan                       249
  hg               read             /                                     250
  hg               read             /builds/fishgk                        260
  sed              read             /ws/ak-on-gate/public                 430
  diff             read             /ws/ak-on-gate/public                2592
```

fsrwcount.d

You can count read/write syscall operations by file system and type.

Script

This is similar to `sysfs.d`, but it prints the file system type instead of the process name:

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

/* trace read() variants, but not readlink() or __pthread*() (macosx) */
syscall::read:entry,
syscall::readv:entry,
syscall::pread*:entry,
syscall::*read*nocancel:entry,
syscall::*write*:entry
{
        @[fds[arg0].fi_fs, probefunc, fds[arg0].fi_mount] = count();
}

dtrace:::END
{
        printf("  %-9s  %-16s %-40s %7s\n", "FS", "SYSCALL", "MOUNTPOINT",
            "COUNT");
        printa("  %-9.9s  %-16s %-40s %@7d\n", @);
}
```

Script fsrwcount.d

Example

Here's an example of running `fsrwcount.d` on Solaris:

```
# fsrwcount.d
Tracing... Hit Ctrl-C to end.
^C
  FS          SYSCALL          MOUNTPOINT                                 COUNT
  specfs      write            /                                              1
  nfs4        read             /ws/onnv-tools                                 3
  zfs         read             /builds/bmc                                    5
  nfs4        read             /home/brendan                                 11
  zfs         read             /builds/ahl                                   16
  sockfs      writev           /                                             20
  zfs         write            /builds/brendan                               30
  <none>      read             <none>                                        33
  sockfs      write            /                                             34
  zfs         read             /var                                          88
  sockfs      read             /                                            104
  zfs         read             /builds/fishgk                               133
  nfs4        write            /ws/ak-on-gate/public                        171
  tmpfs       write            /tmp                                         197
  zfs         read             /builds/brendan                              236
  tmpfs       read             /tmp                                         265
  fifofs      write            /                                            457
  fifofs      read             /                                            625
  zfs         read             /                                            809
  nfs4        read             /ws/ak-on-gate/public                       1673
```

During a software build, this has shown that most of the file system syscalls were reads to the NFSv4 share `/ws/ak-on-gate/public`. The busiest ZFS file systems were `/` followed by `/builds/brendan`.

Here's an example of running `fsrwcount.d` on Mac OS X:

```
# fsrwcount.d
Tracing... Hit Ctrl-C to end.
^C
  FS          SYSCALL          MOUNTPOINT                                 COUNT
  devfs       write            dev                                            2
  devfs       write_nocancel   dev                                            2
  <unknown    write_nocancel   <unknown (not a vnode)>                        3
  hfs         write_nocancel   /                                              6
  devfs       read             dev                                            7
  devfs       read_nocancel    dev                                            7
  hfs         write            /                                             18
  <unknown    write            <unknown (not a vnode)>                       54
  hfs         read_nocancel    /                                             55
  <unknown    read             <unknown (not a vnode)>                      134
  hfs         pwrite           /                                            155
  hfs         read             /                                            507
  hfs         pread            /                                           1760
```

This helps explain line 24, which truncated the FS field to nine characters (`%9.9s`). On Mac OS X, `<unknown (not a vnode>)` may be returned, and without the truncation the columns become crooked. These nonvnode operations may be reads and writes to sockets.

fsrwtime.d

The `fsrwtime.d` script measures the time spent in read and write syscalls, with file system information. The results are printed in distribution plots by microsecond.

Script

If averages or sums are desired instead, change the aggregating function on line 20 and the output formatting on line 26:

```
1   #!/usr/sbin/dtrace -Zs
2
3   /* trace read() variants, but not readlink() or __pthread*() (macosx) */
4   syscall::read:entry,
5   syscall::readv:entry,
6   syscall::pread*:entry,
7   syscall::*read*nocancel:entry,
8   syscall::*write*:entry
9   {
10          self->fd = arg0;
11          self->start = timestamp;
12  }
13
14  syscall::*read*:return,
15  syscall::*write*:return
16  /self->start/
17  {
18          this->delta = (timestamp - self->start) / 1000;
19          @[fds[self->fd].fi_fs, probefunc, fds[self->fd].fi_mount] =
20              quantize(this->delta);
21          self->fd = 0; self->start = 0;
22  }
23
24  dtrace:::END
25  {
26          printa("\n  %s %s (us) \t%s%@d", @);
27  }
```

Script fsrwtime.d

The syscall return probes on lines 14 and 15 use more wildcards without fear of matching unwanted syscalls (such as `readlink()`), since it also checks for `self->start` to be set in the predicate, which will be true only for the syscalls that matched the precise set on lines 4 to 8.

Example

This output shows that `/builds/brendan`, a ZFS file system, mostly returned reads between 8 us and 127 us. These are likely to have returned from the ZFS file system cache, the ARC. The single read that took more than 32 ms is likely to have been returned from disk. More DTracing can confirm.

```
# fsrwtime.d
dtrace: script 'fsrwtime.d' matched 18 probes
^C
CPU     ID                    FUNCTION:NAME
  8      2                             :END
  specfs read (us)      /devices
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              16 |                                         0
[...]

  zfs write (us)        /builds/brendan
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@                                    4
              32 |@@@@@@@@@@@@@@                           11
              64 |@@@@@@@@@@@@@@@@@@@@@                    17
             128 |                                         0

  zfs read (us)         /builds/brendan
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@                         72
              16 |@@@@@@@@@@                               44
              32 |@@@@@@@                                  32
              64 |@@@@@                                    24
             128 |                                         0
             256 |@                                        3
             512 |                                         1
            1024 |                                         0
            2048 |                                         0
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         1
           65536 |                                         0
```

fsrtpk.d

As an example of a different way to analyze time, the `fsrtpk.d` script shows file system read *time per kilobyte*.

Script

This is similar to the `fsrwtime.d` script, but here we divide the time by the number of kilobytes, as read from `arg0` (rval) on read return:

```
#!/usr/sbin/dtrace -Zs

/* trace read() variants, but not readlink() or __pthread*() (macosx) */
syscall::read:entry,
syscall::readv:entry,
syscall::pread*:entry,
syscall::*read*nocancel:entry
{
        self->fd = arg0;
        self->start = timestamp;
}
```

```

syscall::*read*:return
/self->start && arg0 > 0/
{
        this->kb = (arg1 / 1024) ? arg1 / 1024 : 1;
        this->ns_per_kb = (timestamp - self->start) / this->kb;
        @[fds[self->fd].fi_fs, probefunc, fds[self->fd].fi_mount] =
            quantize(this->ns_per_kb);
}

syscall::*read*:return
{
        self->fd = 0; self->start = 0;
}

dtrace:::END
{
        printa("\n  %s %s (ns per kb) \t%s%@d", @);
}
```

Script fsrtpk.d

Example

For the same interval, compare `fsrwtime.d` and `fsrtpk.d`:

```
# fsrwtime.d
[...]
  zfs read (us)         /export/fs1
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |                                         7
               2 |                                         63
               4 |                                         10
               8 |                                         15
              16 |@                                        3141
              32 |@@@@@@                                   27739
              64 |@@@@@@@@@@@                              55730
             128 |@@@@@@@@                                 39625
             256 |@@@@@@@                                  34358
             512 |@@@@                                     18700
            1024 |@@                                       8514
            2048 |@@                                       8407
            4096 |                                         361
            8192 |                                         32
           16384 |                                         1
           32768 |                                         0

# fsrtpk.d
[...]
  zfs read (ns per kb)  /export/fs1
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@@@@@                   109467
             512 |@@@@@@@@@@@@@@@@                         79390
            1024 |@@                                       7643
            2048 |                                         106
            4096 |                                         2
            8192 |                                         0
```

From `fstime.d`, the reads to zfs are quite varied, mostly falling between 32 us and 1024 us. The reason was not varying ZFS performance but varying requested I/O sizes to cached files: Larger I/O sizes take longer to complete because of the movement of data bytes in memory.

The read time per kilobyte is much more consistent, regardless of the I/O size, returning between 256 ns and 1023 ns per kilobyte read.

rwsnoop

The `rwsnoop` script traces `read()` and `write()` syscalls across the system, printing process and size details as they occur. Since these are usually frequent syscalls, the output can be verbose and also prone to feedback loops (this is because the lines of output from `dtrace(1M)` are performed using `write()`, which are also traced by DTrace, triggering more output lines, and so on). The `-n` option can be used to avoid this, allowing process names of interest to be specified.

These syscalls are generic and not exclusively for file system I/O; check the `FILE` column in the output of the script for those that are reading and writing to files.

Script

Since most of this 234-line script handles command-line options, the only interesting DTrace parts are included here. The full script is in the DTraceToolkit and can also be found in `/usr/bin/rwsnoop` on Mac OS X.

The script saves various details in thread-local variables. Here the direction and size of `read()` calls are saved:

```
182   syscall::*read:return
183   /self->ok/
184   {
185           self->rw = "R";
186           self->size = arg0;
187   }
```

which it then prints later:

```
202   syscall::*read:return,
203   syscall::*write:entry
[...]
225           printf("%5d %6d %-12.12s %1s %7d %s\n",
226               uid, pid, execname, self->rw, (int)self->size, self->vpath);
```

This is straightforward. What's not straightforward is the way the file path name is fetched from the file descriptor saved in `self->fd` (line 211):

```
syscall::*read:return,
syscall::*write:entry
/self->ok/
{
        /*
         * Fetch filename
         */
        this->filistp = curthread->t_procp->p_user.u_finfo.fi_list;
        this->ufentryp = (uf_entry_t *)((uint64_t)this->filistp +
            (uint64_t)self->fd * (uint64_t)sizeof(uf_entry_t));
        this->filep = this->ufentryp->uf_file;
        this->vnodep = this->filep != 0 ? this->filep->f_vnode : 0;
        self->vpath = this->vnodep ? (this->vnodep->v_path != 0 ?
            cleanpath(this->vnodep->v_path) : "<unknown>") : "<unknown>";
```

This lump of code digs out the path name from the Solaris kernel and was written this way because `rwsnoop` predates the `fds` array being available in Solaris. With the availability of the `fds[]` array, that entire block of code can be written as follows:

```
self->vpath = fds[self->fd].fi_pathname
```

unless you are using a version of DTrace that doesn't yet have the `fds` array, such as FreeBSD, in which case you can try writing the FreeBSD version of the previous code block.

Examples

The following examples demonstrate the use of the `rwsnoop` script.

Usage: rwsnoop.d.

```
# rwsnoop -h
USAGE: rwsnoop [-hjPtvZ] [-n name] [-p pid]

                -j       # print project ID
                -P       # print parent process ID
                -t       # print timestamp, us
                -v       # print time, string
                -Z       # print zone ID
                -n name  # this process name only
                -p PID   # this PID only
   eg,
        rwsnoop          # default output
        rwsnoop -Z       # print zone ID
        rwsnoop -n bash  # monitor processes named "bash"
```

Web Server. Here `rwsnoop` is used to trace all Web server processes named `httpd` (something that PID-based tools such as `truss(1M)` or `strace` cannot do easily):

```
# rwsnoop -tn httpd
TIME               UID    PID CMD          D   BYTES FILE
6854075939432       80 713149 httpd        R     495 <unknown>
6854075944873       80 713149 httpd        R     495 /wiki/includes/WebResponse.php
6854075944905       80 713149 httpd        R       0 /wiki/includes/WebResponse.php
6854075944921       80 713149 httpd        R       0 /wiki/includes/WebResponse.php
6854075946102       80 713149 httpd        W     100 <unknown>
6854075946261       80 713149 httpd        R     303 <unknown>
6854075946592       80 713149 httpd        W       5 <unknown>
6854075959169       80 713149 httpd        W      92 /var/apache2/2.2/logs/access_log
6854076038294       80 713149 httpd        R       0 <unknown>
6854076038390       80 713149 httpd        R      -1 <unknown>
6854206429906       80 713251 httpd        R    4362 /wiki/includes/LinkBatch.php
6854206429933       80 713251 httpd        R       0 /wiki/includes/LinkBatch.php
6854206429952       80 713251 httpd        R       0 /wiki/includes/LinkBatch.php
6854206432875       80 713251 httpd        W      92 <unknown>
6854206433300       80 713251 httpd        R      52 <unknown>
6854206434656       80 713251 httpd        R    6267 /wiki/includes/SiteStats.php
[...]
```

The files that `httpd` is reading can be seen in the output, along with the log file it is writing to. The `<unknown>` file I/O is likely to be the socket I/O for HTTP, because it reads requests and responds to clients.

mmap.d

Although many of the scripts in this chapter examine file system I/O by tracing reads and writes, there is another way to read or write file data: `mmap()`. This system call maps a region of a file to the memory of the user-land process, allowing reads and writes to be performed by reading and writing to that memory segment. The `mmap.d` script traces mmap calls with details including the process name, filename, and flags used with `mmap()`.

Script

This script was written for Oracle Solaris and uses the preprocessor (`-C` on line 1) so that the `sys/mman.h` file can be included (line 3):

```
1      #!/usr/sbin/dtrace -Cs
2
3      #include <sys/mman.h>
4
5      #pragma D option quiet
6      #pragma D option switchrate=10hz
7
8      dtrace:::BEGIN
9      {
10           printf("%6s %-12s %-4s %-8s %-8s %-8s %s\n", "PID",
11               "PROCESS", "PROT", "FLAGS", "OFFS(KB)", "SIZE(KB)", "PATH");
12      }
13
14     syscall::mmap*:entry
15     /fds[arg4].fi_pathname != "<none>"/
```

```
{
        /* see mmap(2) and /usr/include/sys/mman.h */
        printf("%6d %-12.12s %s%s%s  %s%s%s%s%s%s%s%s %-8d %-8d %s\n",
            pid, execname,
            arg2 & PROT_EXEC  ? "E" : "-",  /* pages can be executed */
            arg2 & PROT_WRITE ? "W" : "-",  /* pages can be written */
            arg2 & PROT_READ  ? "R" : "-",  /* pages can be read */
            arg3 & MAP_INITDATA  ? "I" : "-",     /* map data segment */
            arg3 & MAP_TEXT      ? "T" : "-",     /* map code segment */
            arg3 & MAP_ALIGN     ? "L" : "-",     /* addr specifies alignment */
            arg3 & MAP_ANON      ? "A" : "-",     /* map anon pages directly */
            arg3 & MAP_NORESERVE ? "N" : "-",     /* don't reserve swap area */
            arg3 & MAP_FIXED     ? "F" : "-",     /* user assigns address */
            arg3 & MAP_PRIVATE   ? "P" : "-",     /* changes are private */
            arg3 & MAP_SHARED    ? "S" : "-",     /* share changes */
            arg5 / 1024, arg1 / 1024, fds[arg4].fi_pathname);
}
```

Script mmap.d

Example

While tracing, the `cp(1)` was executed to copy a 100MB file called `100m`:

```
solaris# cp /export/fs1/100m /export/fs2
```

The file was read by `cp(1)` by mapping it to memory, 8MB at a time:

```
solaris# mmap.d
   PID PROCESS       PROT FLAGS     OFFS(KB) SIZE(KB) PATH
  2652 cp            E-R  --L---P- 0        32       /lib/libc.so.1
  2652 cp            E-R  -T---FP- 0        1274     /lib/libc.so.1
  2652 cp            EWR  I----FP- 1276     27       /lib/libc.so.1
  2652 cp            E-R  --L---P- 0        32       /lib/libsec.so.1
  2652 cp            E-R  -T---FP- 0        62       /lib/libsec.so.1
  2652 cp            -WR  I----FP- 64       15       /lib/libsec.so.1
  2652 cp            E-R  --L---P- 0        32       /lib/libcmdutils.so.1
  2652 cp            E-R  -T---FP- 0        11       /lib/libcmdutils.so.1
  2652 cp            -WR  I----FP- 12       0        /lib/libcmdutils.so.1
  2652 cp            --R  -------S 0        8192     /export/fs1/100m
  2652 cp            --R  -----F-S 8192     8192     /export/fs1/100m
  2652 cp            --R  -----F-S 16384    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 24576    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 32768    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 40960    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 49152    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 57344    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 65536    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 73728    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 81920    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 90112    8192     /export/fs1/100m
  2652 cp            --R  -----F-S 98304    4096     /export/fs1/100m
^C
```

The output also shows the initialization of the `cp(1)` command because it maps libraries as executable segments.

fserrors.d

Errors can be particularly interesting when troubleshooting system issues, including errors returned by the file system in response to application requests. This script traces all errors at the syscall layer, providing process, path name, and error number information. Many of these errors may be "normal" for the application and handled correctly by the application code. This script merely reports that they happened, not how they were then handled (if they were handled).

Script

This script traces variants of `read()`, `write()`, `open()`, and `stat()`, which are handled a little differently depending on how to retrieve the path information. It can be enhanced to include other file system system calls as desired:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       trace("Tracing syscall errors... Hit Ctrl-C to end.\n");
}

syscall::read*:entry, syscall::write*:entry { self->fd = arg0; }
syscall::open*:entry, syscall::stat*:entry  { self->ptr = arg0; }

syscall::read*:return, syscall::write*:return
/(int)arg0 < 0 && self->fd > 2/
{
        self->path = fds[self->fd].fi_pathname;
}

syscall::open*:return, syscall::stat*:return
/(int)arg0 < 0 && self->ptr/
{
        self->path = copyinstr(self->ptr);
}

syscall::read*:return, syscall::write*:return,
syscall::open*:return, syscall::stat*:return
/(int)arg0 < 0 && self->path != NULL/
{
        @[execname, probefunc, errno, self->path] = count();
        self->path = 0;
}

syscall::read*:return, syscall::write*:return { self->fd = 0; }
syscall::open*:return, syscall::stat*:return  { self->ptr = 0; }

dtrace:::END
{
        printf("%16s %16s %3s %8s %s\n", "PROCESSES", "SYSCALL", "ERR",
            "COUNT", "PATH");
        printa("%16s %16s %3d %@8d %s\n", @);
}
```

Script fserrors.d

Example

`fserrors.d` was run for one minute on a wiki server (running both TWiki and MediaWiki):

```
# fserrors.d
       PROCESSES          SYSCALL ERR    COUNT PATH
            sshd             open   2        1 /etc/hosts.allow
            sshd             open   2        1 /etc/hosts.deny
[...output truncated...]
            sshd           stat64   2        2 /root/.ssh/authorized_keys
            sshd           stat64   2        2 /root/.ssh/authorized_keys2
          locale             open   2        4 /var/ld/ld.config
            sshd             open   2        5 /var/run/tzsync
            view           stat64   2        7 /usr/local/twiki/data/Main/NFS.txt
            view           stat64   2        8 /usr/local/twiki/data/Main/ARC.txt
            view           stat64   2       11 /usr/local/twiki/data/Main/TCP.txt
            Xorg             read  11       27 <unknown>
            view           stat64   2       32 /usr/local/twiki/data/Main/NOTES.txt
           httpd             read  11       35 <unknown>
            view           stat64   2       85 /usr/local/twiki/data/Main/DRAM.txt
            view           stat64   2      174 /usr/local/twiki/data/Main/ZFS.txt
            view           stat64   2      319 /usr/local/twiki/data/Main/IOPS.txt
```

While tracing, processes with the name `view` attempted to `stat64()` an `IOPS.txt` file 319 times, each time encountering error number 2 (file not found). The `view` program was short-lived and not still running on the system and so was located by using a DTrace one-liner to catch its execution:

```
# dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
dtrace: description 'proc:::exec-success ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2  23001     exec_common:exec-success   /usr/bin/perl -wT /usr/local/twiki/bin/view
```

It took a little more investigation to find the reason behind the `stat64()` calls: TWiki automatically detects terms in documentation by searching for words in all capital letters and then checks whether there are pages for those terms. Since TWiki saves everything as text files, it checks by running `stat64()` on the file system for those pages (indirectly, since it is a Perl program). If this sounds suboptimal, use DTrace to measure the CPU time spent calling `stat64()` to quantify this behavior—`stat()` is typically a fast call.

fsinfo Scripts

The fsinfo provider traces file system activity at the VFS layer, allowing all file system activity to be traced within the kernel from one provider. The probes it exports contain mapped file info and byte counts where appropriate. It is currently available only on Solaris; FreeBSD has a similar provider called vfs.

fswho.d

This script uses the fsinfo provider to show which processes are reading and writing to which file systems, in terms of kilobytes.

Script

This is similar to the earlier `sysfs.d` script, but it can match all file system reads and writes without tracing all the syscalls that may be occurring. It can also easily access the size of the reads and writes, provided as `arg1` by the fsinfo provider (which isn't always easy at the syscall provider: Consider `readv()`).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fsinfo:::read,
fsinfo:::write
{
        @[execname, probename == "read" ? "R" : "W", args[0]->fi_fs,
            args[0]->fi_mount] = sum(arg1);
}

dtrace:::END
{
        normalize(@, 1024);
        printf("  %-16s  %1s %12s  %-10s %s\n", "PROCESSES", "D", "KBYTES",
            "FS", "MOUNTPOINT");
        printa("  %-16s  %1.1s %@12d  %-10s %s\n", @);
}
```

Script fswho.d

Example

The source code was building on a ZFS share while `fswho.d` was run:

```
# fswho.d
Tracing... Hit Ctrl-C to end.
^C
  PROCESSES         D       KBYTES  FS          MOUNTPOINT
  tail              R            0  zfs         /builds/ahl
  tail              R            0  zfs         /builds/bmc
  sshd              R            0  sockfs      /
  sshd              W            0  sockfs      /
  ssh-socks5-proxy  R            0  sockfs      /
```

```
  sh                W            1  tmpfs      /tmp
  dmake             R            1  nfs4       /home/brendan
[...output truncated...]
  id                R           68  zfs        /var
  cp                R          133  zfs        /builds/brendan
  scp               R          224  nfs4       /net/fw/export/install
  install           R          289  zfs        /
  dmake             R          986  zfs        /
  cp                W         1722  zfs        /builds/brendan
  dmake             W        13357  zfs        /builds/brendan
  dmake             R        21820  zfs        /builds/brendan
```

`fswho.d` has identified that processes named `dmake` read 21MB from the `/builds/brendan` share and wrote back 13MB. Various other process file system activity has also been identified, which includes socket I/O because the kernel implementation serves these via a sockfs file system.

readtype.d

This script shows the type of reads by file system and the amount for comparison, differentiating between logical reads (syscall layer) and physical reads (disk layer). There are a number of reasons why the rate of logical reads will not equal physical reads.

Caching: Logical reads may return from a DRAM cache without needing to be satisfied as physical reads from the storage devices.

Read-ahead/prefetch: The file system may detect a sequential access pattern and request data to prewarm the cache before it has been requested logically. If it is then never requested logically, more physical reads may occur than logical.

File system record size: The file system on-disk structure may store data as addressable blocks of a certain size (record size), and physical reads to storage devices will be in units of this size. This may inflate reads between logical and physical, because they are rounded up to record-sized reads for the physical storage devices.

Device sector size: Despite the file system record size, there may still be a minimum physical read size required by the storage device, such as 512 bytes for common disk drives (sector size).

As an example of file system record size inflation, consider a file system that employs a fixed 4KB record size, while an application is performing random 512-byte reads. Each logical read will be 512 bytes in size, but each physical read will be 4KB—reading an extra 3.5KB that will not be used (or is unlikely to be used, because the workload is random). This makes for an 8x inflation between logical and physical reads.

Script

This script uses the fsinfo provider to trace logical reads and uses the io provider to trace physical reads. It is based on `rfsio.d` from the DTraceToolkit.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

inline int TOP = 20;
self int trace;
uint64_t lbytes;
uint64_t pbytes;

dtrace:::BEGIN
{
        trace("Tracing... Output every 5 secs, or Ctrl-C.\n");
}

fsinfo:::read
{
        @io[args[0]->fi_mount, "logical"] = count();
        @bytes[args[0]->fi_mount, "logical"] = sum(arg1);
        lbytes += arg1;
}

io:::start
/args[0]->b_flags & B_READ/
{
        @io[args[2]->fi_mount, "physical"] = count();
        @bytes[args[2]->fi_mount, "physical"] = sum(args[0]->b_bcount);
        pbytes += args[0]->b_bcount;
}

profile:::tick-5s,
dtrace:::END
{
        trunc(@io, TOP);
        trunc(@bytes, TOP);
        printf("\n%Y:\n", walltimestamp);
        printf("\n Read I/O (top %d)\n", TOP);
        printa(" %-32s %10s %10@d\n", @io);
        printf("\n Read Bytes (top %d)\n", TOP);
        printa(" %-32s %10s %10@d\n", @bytes);
        printf("\nphysical/logical bytes rate: %d%%\n",
            lbytes ? 100 * pbytes / lbytes : 0);
        trunc(@bytes);
        trunc(@io);
        lbytes = pbytes = 0;
}
```

Script readtype.d

Examples

Examples include uncached file system read and cache file system read.

Uncached File System Read. Here the `/usr` file system is archived, reading through the files sequentially:

```
# readtype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:42:50:

 Read I/O (top 20)
 /                                  logical        13
 /export/home                       logical        23
 /tmp                               logical       428
 /usr                              physical      1463
 /usr                               logical      2993

 Read Bytes (top 20)
 /tmp                               logical         0
 /                                  logical      1032
 /export/home                       logical     70590
 /usr                               logical  11569675
 /usr                              physical  11668480

physical/logical bytes rate: 102%
```

The physical/logical throughput rate was 102 percent during this interval. The reasons for the inflation may be because of both sector size (especially when reading any file smaller than 512 bytes) and read-ahead (where tracing has caught the physical but not yet the logical reads).

Cache File System Read. Following on from the previous example, the `/usr` file system was reread:

```
# readtype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:44:05:

 Read I/O (top 20)
 /                                 physical         5
 /                                  logical        21
 /export/home                       logical        54
 /tmp                               logical       865
 /usr                              physical      3005
 /usr                               logical     14029

 Read Bytes (top 20)
 /tmp                               logical         0
 /                                  logical      1372
 /                                 physical     24576
 /export/home                       logical    166561
 /usr                              physical  16015360
 /usr                               logical  56982746

physical/logical bytes rate: 27%
```

Now much of data is returning from the cache, with only 27 percent being read from disk. We can see the difference this makes to the application: The first example showed a logical read throughput of 11MB during the five-second interval as the data was read from disk; the logical rate in this example is now 56MB during five seconds.

writetype.d

As a companion to `readtype.d`, this script traces file system writes, allowing types to be compared. Logical writes may differ from physical writes for the following reasons (among others):

- **Asynchronous writes**: The default behavior[1] for many file systems is that logical writes dirty data in DRAM, which is later flushed to disk by an asynchronous thread. This allows the application to continue without waiting for the disk writes to complete. The effect seen in `writetype.d` will be logical writes followed some time later by physical writes.
- **Write canceling**: Data logically written but not yet physically written to disk is logically overwritten, canceling the previous physical write.
- **File system record size**: As described earlier for `readtype.d`.
- **Device sector size**: As described earlier for `readtype.d`.
- **Volume manager**: If software volume management is used, such as applying levels of RAID, writes may be inflated depending on the RAID configuration. For example, software mirroring will cause logical writes to be doubled when they become physical.

Script

This script is identical to `readtype.d` except for the following lines:

```
15  fsinfo:::write

22  io:::start
23  /!(args[0]->b_flags & B_READ)/

36          printf("\n Write I/O (top %d)\n", TOP);

38          printf("\n Write Bytes (top %d)\n", TOP);
```

Now fsinfo is tracing writes, and the `io:::start` predicate also matches writes.

1. For times when the application requires the data to be written on stable storage before continuing, `open()` flags such as `O_SYNC` and `O_DSYNC` can be used to inform the file system to write immediately to stable storage.

Example

The `writetype.d` script was run for ten seconds. During the first five seconds, an application wrote data to the file system:

```
# writetype.d
Tracing... Output every 5 secs, or Ctrl-C.

2010 Jun 19 07:59:10:

 Write I/O (top 20)
 /var                                logical          1
 /                                   logical          3
 /export/ufs1                        logical          9
 /export/ufs1                       physical        696

 Write bytes (top 20)
 /                                   logical        208
 /var                                logical        704
 /export/ufs1                       physical    2587648
 /export/ufs1                        logical    9437184

physical/logical throughput rate: 24%

2010 Jun 19 07:59:15:

 Write I/O (top 20)
 /                                   logical          2
 /export/ufs1                       physical        238

 Write bytes (top 20)
 /                                   logical        752
 /export/ufs1                       physical    7720960

physical/logical throughput rate: 805%
```

In the first five-second summary, more logical bytes were written than physical, because writes were buffered in the file system cache but not yet flushed to disk. The second output shows those writes finishing being flushed to disk.

fssnoop.d

This script traces all file system activity by printing every event from the fsinfo provider with user, process, and size information, as well as path information if available. It also prints all the event data line by line, without trying to summarize it into reports, making the output suitable for other postprocessing if desired. The section that follows demonstrates rewriting this script for other providers and operating systems.

Script

Since this traces all file system activity, it may catch sockfs activity and create a feedback loop where the DTrace output to the file system or your remote network session is traced. To work around this, it accepts an optional argument of the process name to trace and excludes dtrace processes by default (line 14). For more sophisticated arguments, the script could be wrapped in the shell like `rwsnoop` so that `getopts` can be used.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option defaultargs
5   #pragma D option switchrate=10hz
6
7   dtrace:::BEGIN
8   {
9          printf("%-12s %6s %6s %-12.12s %-12s %-6s %s\n", "TIME(ms)", "UID",
10              "PID", "PROCESS", "CALL", "BYTES", "PATH");
11  }
12
13  fsinfo:::
14  /execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
15  {
16          printf("%-12d %6d %6d %-12.12s %-12s %-6d %s\n", timestamp / 1000000,
17              uid, pid, execname, probename, arg1, args[0]->fi_pathname);
18  }
```

Script fssnoop.d

So that the string argument `$$1` could be optional, line 4 sets the `defaultargs` option, which sets `$$1` to `NULL` if it wasn't provided at the command line. Without `defaultargs`, DTrace would error unless an argument is provided.

Examples

The default output prints all activity:

```
# fssnoop.d
TIME(ms)         UID    PID PROCESS      CALL         BYTES  PATH
924434524          0   2687 sshd         poll         0      <unknown>
924434524          0   2687 sshd         rwlock       0      <unknown>
924434524          0   2687 sshd         write        112    <unknown>
924434524          0   2687 sshd         rwunlock     0      <unknown>
[...]
```

Since it was run over an SSH session, it sees its own socket writes to sockfs by the sshd process. An output file can be specified to prevent this:

```
# fssnoop.d -o out.log
# cat out.log
TIME(ms)        UID    PID PROCESS      CALL         BYTES  PATH
924667432         0   7108 svcs         lookup       0      /usr/share/lib/zoneinfo
924667432         0   7108 svcs         lookup       0      /usr/share/lib/zoneinfo/UTC
```

```
924667432         0   7108 svcs          getattr     0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          access      0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          open        0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          getattr     0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          rwlock      0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          read        56     /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          rwunlock    0      /usr/share/lib/zoneinfo/UTC
924667432         0   7108 svcs          close       0      /usr/share/lib/zoneinfo/UTC
[...]
```

This has caught the execution of the Oracle Solaris `svcs(1)` command, which was listing system services. The UTC file was read in this way 204 times (the output was many pages long), which is twice for every line of output that `svcs(1)` printed, which included a date.

To filter on a particular process name, you can provided as an argument. Here, the file system calls from the `ls(1)` command were traced:

```
# fssnoop.d ls
TIME(ms)        UID    PID PROCESS       CALL         BYTES  PATH
924727221         0   7111 ls            rwlock       0      /tmp
924727221         0   7111 ls            readdir      1416   /tmp
924727221         0   7111 ls            rwunlock     0      /tmp
924727221         0   7111 ls            rwlock       0      /tmp
[...]
```

VFS Scripts

VFS is the Virtual File System, a kernel interface that allows different file systems to integrate into the same kernel code. It provides an abstraction of a file system with the common calls: read, write, open, close, and so on. Interfaces and abstractions can make good targets for DTracing, since they are often documented and relatively stable (compared to the implementation code).

The fsinfo provider for Solaris traces at the VFS level, as shown by the scripts in the previous "fsinfo" section. FreeBSD has the vfs provider for this purpose, demonstrated in this section. When neither vfs or fsinfo is available, VFS can be traced using the fbt[2] provider. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and on Mac OS X version 10.6, and they may not work on other releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for VFS analysis.

2. See the "fbt Provider" section in Chapter 12 for more discussion about use of the fbt provider.

To demonstrate the different ways VFS can be traced and to allow these to be compared, the `fssnoop.d` script has been written in four ways:

- `fssnoop.d`: fsinfo provider based (OpenSolaris), shown previously
- `solvfssnoop.d`: fbt provider based (Solaris)
- `macvfssnoop.d`: fbt provider based (Mac OS X)
- `vfssnoop.d`: vfs provider based (FreeBSD)

Because these scripts trace common VFS events, they can be used as starting points for developing other scripts. This section also includes three examples that trace file creation and deletion on the different operating systems (`sollife.d`, `maclife.d`, and `vfslife.d`).

Note that VFS can cover more than just on-disk file systems; whichever kernel modules use the VFS abstraction may also be traced by these scripts, including terminal output (writes to `/dev/pts` or `dev/tty` device files).

solvfssnoop.d

To trace VFS calls in the Oracle Solaris kernel, the fop interface can be traced using the fbt provider. (This is also the location that the fsinfo provider instruments.) Here's an example of listing `fop` probes:

```
solaris# dtrace -ln 'fbt::fop_*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
36831        fbt           genunix                      fop_inactive entry
38019        fbt           genunix                        fop_addmap entry
38023        fbt           genunix                        fop_access entry
38150        fbt           genunix                        fop_create entry
38162        fbt           genunix                        fop_delmap entry
38318        fbt           genunix                        fop_frlock entry
38538        fbt           genunix                        fop_lookup entry
38646        fbt           genunix                         fop_close entry
[...output truncated...]
```

The function names include the names of the VFS calls. Although the fbt provider is considered an unstable interface, tracing kernel interfaces such as `fop` is expected to be the safest use of fbt possible—`fop` doesn't change much (but be aware that it can and has).

Script

This script traces many of the common VFS calls at the Oracle Solaris `fop` interface, including `read()`, `write()` and `open()`. See `/usr/include/sys/vnode.h` for the full list. Additional calls can be added to `solvfssnoop.d` as desired.

```
1     #!/usr/sbin/dtrace -s
2
3     #pragma D option quiet
4     #pragma D option defaultargs
5     #pragma D option switchrate=10hz
6
7     dtrace:::BEGIN
8     {
9           printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
10              "PID", "PROCESS", "CALL", "KB", "PATH");
11    }
12
13    /* see /usr/include/sys/vnode.h */
14
15    fbt::fop_read:entry, fbt::fop_write:entry
16    {
17          self->path = args[0]->v_path;
18          self->kb = args[1]->uio_resid / 1024;
19    }
20
21    fbt::fop_open:entry
22    {
23          self->path = (*args[0])->v_path;
24          self->kb = 0;
25    }
26
27    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
28    fbt::fop_readdir:entry
29    {
30          self->path = args[0]->v_path;
31          self->kb = 0;
32    }
33
34    fbt::fop_read:entry, fbt::fop_write:entry, fbt::fop_open:entry,
35    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
36    fbt::fop_readdir:entry
37    /execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
38    {
39          printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
40              uid, pid, execname, probefunc, self->kb,
41              self->path != NULL ? stringof(self->path) : "<null>");
42    }
43
44    fbt::fop_read:entry, fbt::fop_write:entry, fbt::fop_open:entry,
45    fbt::fop_close:entry, fbt::fop_ioctl:entry, fbt::fop_getattr:entry,
46    fbt::fop_readdir:entry
47    {
48          self->path = 0; self->kb = 0;
49    }
```

Script solvfssnoop.d

Lines 15 to 32 probe different functions and populate the `self->path` and `self->kb` variables so that they are printed out in a common block of code on lines 39 to 41.

Example

As with `fssnoop.d`, this script accepts an optional argument for the process name to trace. Here's an example of tracing `ls -l`:

```
solaris# solvfssnoop.d ls
TIME(ms)       UID    PID PROCESS     CALL         KB   PATH
2499844          0   1152 ls          fop_close    0    /var/run/name_service_door
2499844          0   1152 ls          fop_close    0    <null>
2499844          0   1152 ls          fop_close    0    /dev/pts/2
2499844          0   1152 ls          fop_getattr  0    /usr/bin/ls
2499844          0   1152 ls          fop_getattr  0    /lib/libc.so.1
2499844          0   1152 ls          fop_getattr  0    /usr/lib/libc/libc_hwcap1.so.1
2499844          0   1152 ls          fop_getattr  0    /lib/libc.so.1
2499844          0   1152 ls          fop_getattr  0    /usr/lib/libc/libc_hwcap1.so.1
[...]
2499851          0   1152 ls          fop_getattr  0    /var/tmp
2499851          0   1152 ls          fop_open     0    /var/tmp
2499851          0   1152 ls          fop_getattr  0    /var/tmp
2499852          0   1152 ls          fop_readdir  0    /var/tmp
2499852          0   1152 ls          fop_getattr  0    /var/tmp/ExrUaWjc
[...]
2500015          0   1152 ls          fop_open     0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_getattr  0    /etc/passwd
2500015          0   1152 ls          fop_ioctl    0    /etc/passwd
2500015          0   1152 ls          fop_read     1    /etc/passwd
2500016          0   1152 ls          fop_getattr  0    /etc/passwd
2500016          0   1152 ls          fop_close    0    /etc/passwd
[...]
```

The output has been truncated to highlight three stages of `ls` that can be seen in the VFS calls: command initialization, reading the directory, and reading system databases.

macvfssnoop.d

To trace VFS calls in the Mac OS X kernel, the VNOP interface can be traced using the fbt provider. Here's an example of listing VNOP probes:

```
macosx# dtrace -ln 'fbt::VNOP_*:entry'
ID   PROVIDER            MODULE                          FUNCTION NAME
  705        fbt       mach_kernel                         VNOP_ACCESS entry
  707        fbt       mach_kernel                        VNOP_ADVLOCK entry
  709        fbt       mach_kernel                       VNOP_ALLOCATE entry
  711        fbt       mach_kernel                       VNOP_BLKTOOFF entry
  713        fbt       mach_kernel                       VNOP_BLOCKMAP entry
  715        fbt       mach_kernel                         VNOP_BWRITE entry
  717        fbt       mach_kernel                          VNOP_CLOSE entry
  719        fbt       mach_kernel                       VNOP_COPYFILE entry
  721        fbt       mach_kernel                         VNOP_CREATE entry
  723        fbt       mach_kernel                       VNOP_EXCHANGE entry
  725        fbt       mach_kernel                          VNOP_FSYNC entry
  727        fbt       mach_kernel                        VNOP_GETATTR entry
[...output truncated...]
```

The kernel source can be inspected to determine the arguments to these calls.

Script

This script traces many of the common VFS calls at the Darwin VNOP interface, including `read()`, `write()`, and `open()`. See `sys/bsd/sys/vnode_if.h` from the source for the full list. Additional calls can be added as desired.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "KB", "PATH");
}

/* see sys/bsd/sys/vnode_if.h */

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = ((struct uio *)arg1)->uio_resid_64 / 1024;
}

fbt::VNOP_OPEN:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = 0;
}

fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
{
      self->path = ((struct vnode *)arg0)->v_name;
      self->kb = 0;
}

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry, fbt::VNOP_OPEN:entry,
fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
/execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
{
      printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
          uid, pid, execname, probefunc, self->kb,
          self->path != NULL ? stringof(self->path) : "<null>");
}

fbt::VNOP_READ:entry, fbt::VNOP_WRITE:entry, fbt::VNOP_OPEN:entry,
fbt::VNOP_CLOSE:entry, fbt::VNOP_IOCTL:entry, fbt::VNOP_GETATTR:entry,
fbt::VNOP_READDIR:entry
{
      self->path = 0; self->kb = 0;
}
```

Script macvfssnoop.d

Example

An `ls -l` command was traced to compare with the other VFS script examples:

```
macosx# macvfssnoop.d ls
TIME(ms)         UID    PID PROCESS      CALL         KB   PATH
1183135202       501  57611 ls           VNOP_GETATTR 0    urandom
1183135202       501  57611 ls           VNOP_OPEN    0    urandom
1183135202       501  57611 ls           VNOP_READ    0    urandom
1183135202       501  57611 ls           VNOP_CLOSE   0    urandom
1183135202       501  57611 ls           VNOP_GETATTR 0    libncurses.5.4.dylib
1183135202       501  57611 ls           VNOP_GETATTR 0    libSystem.B.dylib
1183135202       501  57611 ls           VNOP_GETATTR 0    libSystem.B.dylib
1183135202       501  57611 ls           VNOP_GETATTR 0    libmathCommon.A.dylib
1183135203       501  57611 ls           VNOP_GETATTR 0    libmathCommon.A.dylib
[…]
1183135221       501  57611 ls           VNOP_GETATTR 0    fswho
1183135221       501  57611 ls           VNOP_GETATTR 0    macvfssnoop.d
1183135221       501  57611 ls           VNOP_GETATTR 0    macvfssnoop.d
1183135221       501  57611 ls           VNOP_GETATTR 0    new
1183135221       501  57611 ls           VNOP_GETATTR 0    oneliners
[…]
1183135225       501  57611 ls           VNOP_GETATTR 0    fswho
1183135225       501  57611 ls           VNOP_WRITE   0    ttys003
1183135225       501  57611 ls           VNOP_GETATTR 0    macvfssnoop.d
1183135225       501  57611 ls           VNOP_GETATTR 0    macvfssnoop.d
1183135225       501  57611 ls           VNOP_WRITE   0    ttys003
[…]
```

The VFS calls show three stages to `ls` on Mac OS X: command initialization, an initial check of the files, and then a second pass as output is written to the screen (`ttys003`).

vfssnoop.d

FreeBSD has the VOP interface for VFS, which is similar to the VNOP interface on Mac OS X (as traced by `macvfssnoop.d`). Instead of tracing VOP via the fbt provider, this script demonstrates the FreeBSD vfs provider.[3] Here's an example listing vfs probes:

```
freebsd# dtrace -ln vfs:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
38030        vfs         namecache                      zap_negative done
38031        vfs         namecache                               zap done
38032        vfs         namecache                          purgevfs done
38033        vfs         namecache                    purge_negative done
38034        vfs         namecache                             purge done
38035        vfs         namecache                            lookup miss
38036        vfs         namecache                            lookup hit_negative
38037        vfs         namecache                            lookup hit
38038        vfs         namecache                          fullpath return
```

3. This was written by Robert Watson.

```
 38039       vfs        namecache                          fullpath miss
 38040       vfs        namecache                          fullpath hit
 38041       vfs        namecache                          fullpath entry
 38042       vfs        namecache                    enter_negative done
 38043       vfs        namecache                             enter done
 38044       vfs            namei                            lookup return
 38045       vfs            namei                            lookup entry
 38046       vfs                                               stat reg
 38047       vfs                                               stat mode
 38048       vfs              vop                       vop_vptocnp return
 38049       vfs              vop                       vop_vptocnp entry
 38050       vfs              vop                        vop_vptofh return
 38051       vfs              vop                        vop_vptofh entry
[...]
```

Four different types of probes are shown in this output:

`vfs:namecache::`: Name cache operations, including lookups (hit/miss)

`vfs:namei::`: Filename to vnode lookups

`vfs::stat::`: Stat calls

`vfs:vop::`: VFS operations

The `vfssnoop.d` script demonstrates three of these (`namecache`, `namei`, and `vop`).

Script

The `vfs:vop::` probes traces VFS calls on vnodes, which this script converts into path names or filenames for printing. On FreeBSD, vnodes don't contain a cached path name and may not contain a filename either unless it's in the `(struct namecache *) v_cache_dd` member. There are a few ways to tackle this; here, vnode to path or filename mappings are cached during `namei()` calls and `namecache` hits, both of which can also be traced from the vfs provider:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz
#pragma D option dynvarsize=4m

dtrace:::BEGIN
{
        printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
            "PID", "PROCESS", "CALL", "KB", "PATH/FILE");
}

/*
 * Populate Vnode2Path from namecache hits
 */
vfs:namecache:lookup:hit
/V2P[arg2] == NULL/
```

continues

```
{
      V2P[arg2] = stringof(arg1);
}

/*
 * (Re)populate Vnode2Path from successful namei() lookups
 */
vfs:namei:lookup:entry
{
      self->buf = arg1;
}
vfs:namei:lookup:return
/self->buf != NULL && arg0 == 0/
{
      V2P[arg1] = stringof(self->buf);
}
vfs:namei:lookup:return
{
      self->buf = 0;
}

/*
 * Trace and print VFS calls
 */
vfs::vop_read:entry, vfs::vop_write:entry
{
      self->path = V2P[arg0];
      self->kb = args[1]->a_uio->uio_resid / 1024;
}


vfs::vop_open:entry, vfs::vop_close:entry, vfs::vop_ioctl:entry,
vfs::vop_getattr:entry, vfs::vop_readdir:entry
{
      self->path = V2P[arg0];
      self->kb = 0;
}

vfs::vop_read:entry, vfs::vop_write:entry, vfs::vop_open:entry,
vfs::vop_close:entry, vfs::vop_ioctl:entry, vfs::vop_getattr:entry,
vfs::vop_readdir:entry
/execname != "dtrace" && ($$1 == NULL || $$1 == execname)/
{
      printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
          uid, pid, execname, probefunc, self->kb,
          self->path != NULL ? self->path : "<unknown>");
}

vfs::vop_read:entry, vfs::vop_write:entry, vfs::vop_open:entry,
vfs::vop_close:entry, vfs::vop_ioctl:entry, vfs::vop_getattr:entry,
vfs::vop_readdir:entry
{
      self->path = 0; self->kb = 0;
}

/*
 * Tidy V2P, otherwise it gets too big (dynvardrops)
 */
vfs:namecache:purge:done,
vfs::vop_close:entry
{
      V2P[arg0] = 0;
}
```

Script vfssnoop.d

The V2P array can get large, and frequent probes events may cause dynamic variable drops. To reduce these drops, the V2P array is trimmed in lines 76 to 80, and the `dynvarsize` tunable is increased on line 6 (but may need to be set higher, depending on your workload).

Example

An `ls -l` command was traced to compare with the other VFS script examples:

```
freebsd# vfssnoop.d ls
TIME(ms)         UID    PID PROCESS      CALL         KB   PATH/FILE
167135998          0  29717 ls           vop_close    0    /bin/ls
167135999          0  29717 ls           vop_open     0    /var/run/ld-elf.so.hints
167135999          0  29717 ls           vop_read     0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_read     0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_close    0    /var/run/ld-elf.so.hints
167136000          0  29717 ls           vop_open     0    /lib/libutil.so.8
[...]
167136007          0  29717 ls           vop_getattr  0    .history
167136007          0  29717 ls           vop_getattr  1    .bash_history
167136008          0  29717 ls           vop_getattr  0    .ssh
167136008          0  29717 ls           vop_getattr  0    namecache.d
167136008          0  29717 ls           vop_getattr  0    vfssnoop.d
[...]
167136011          0  29717 ls           vop_read     0    /etc/spwd.db
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_read     4    /etc/spwd.db
167136011          0  29717 ls           vop_getattr  0    /etc/nsswitch.conf
167136011          0  29717 ls           vop_open     0    /etc/group
[...]
```

The three stages of `ls` shown here are similar to those seen on Oracle Solaris: command initialization, reading the directory, and reading system databases. In some cases, `vfssnoop.d` is able to print full path names; in others, it prints only the filename.

sollife.d

This script shows file creation and deletion events only. It's able to identify file system churn—the rapid creation and deletion of temporary files. Like `solfssnoop.d`, it traces VFS calls using the fbt provider.

Script

This is a reduced version of `solfssnoop.d`, which traces only the `create()` and `remove()` events:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "PATH");
}

/* see /usr/include/sys/vnode.h */

fbt::fop_create:entry,
fbt::fop_remove:entry
{
      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          args[0]->v_path != NULL ? stringof(args[0]->v_path) : "<null>",
          stringof(arg1));
}
```

Script sollife.d

Example

Here the script has caught the events from the `vim(1)` text editor, which opened the script in a different terminal window, made a change, and then saved and quit:

```
# sollife.d
TIME(ms)        UID    PID PROCESS      CALL         PATH
1426193948   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swp
1426193953   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swx
1426193956   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swx
1426193958   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swp
1426193961   130948 112454 vim          fop_create   /home/brendan/.sollife.d.swp
1426205215   130948 112454 vim          fop_create   /home/brendan/4913
1426205230   130948 112454 vim          fop_remove   /home/brendan/4913
1426205235   130948 112454 vim          fop_create   /home/brendan/sollife.d
1426205244   130948 112454 vim          fop_remove   /home/brendan/sollife.d~
1426205246   130948 112454 vim          fop_create   /home/brendan/.viminfz.tmp
1426205256   130948 112454 vim          fop_remove   /home/brendan/.viminfo
1426205262   130948 112454 vim          fop_remove   /home/brendan/.sollife.d.swp
```

The output shows the temporary swap files created and then removed by `vim`. This script could be enhanced to trace `rename()` events as well, which may better explain how `vim` is managing these files.

maclife.d

This is the `sollife.d` script, written for Mac OS X. As with `macvfssnoop.d`, it uses the fbt provider to trace VNOP interface calls:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "DIR/FILE");
}

/* see sys/bsd/sys/vnode_if.h */

fbt::VNOP_CREATE:entry,
fbt::VNOP_REMOVE:entry
{
      this->path = ((struct vnode *)arg0)->v_name;
      this->name = ((struct componentname *)arg2)->cn_nameptr;
      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          this->path != NULL ? stringof(this->path) : "<null>",
          stringof(this->name));
}
```

Script maclife.d

vfslife.d

This is the `sollife.d` script, written for FreeBSD. As with `vfssnoop.d`, it uses the vfs provider. This time it attempts to retrieve a directory name from the directory vnode namecache entry (`v_cache_dd`), instead of using DTrace to cache vnode to path translations.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-12s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "DIR/FILE");
}

/* see sys/bsd/sys/vnode_if.h */

vfs::vop_create:entry,
vfs::vop_remove:entry
{
      this->dir = args[0]->v_cache_dd != NULL ?
          stringof(args[0]->v_cache_dd->nc_name) : "<null>";
      this->name = args[1]->a_cnp->cn_nameptr != NULL ?
          stringof(args[1]->a_cnp->cn_nameptr) : "<null>";

      printf("%-12d %6d %6d %-12.12s %-12s %s/%s\n",
          timestamp / 1000000, uid, pid, execname, probefunc,
          this->dir, this->name);
}
```

Script vfslife.d

dnlcps.d

The Directory Name Lookup Cache is a Solaris kernel facility used to cache path names to vnodes. This script shows its hit rate by process, which can be poor when path names are used that are too long for the DNLC. A similar script can be written for the other operating systems; FreeBSD has the `vfs:namecache:lookup:` probes for this purpose.

Script

```
#!/usr/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

fbt::dnlc_lookup:return
{
        this->code = arg1 == 0 ? 0 : 1;
        @Result[execname, pid] = lquantize(this->code, 0, 1, 1);
}

dtrace:::END
{
        printa(" CMD: %-16s PID: %d\n%@d\n", @Result);
}
```

Script dnlcps.d

Example

The DNLC lookup result is shown in a distribution plot for visual comparison. Here, a `tar(1)` command had a high hit rate (hit == 1) compared to misses.

```
# dnlcps.d
Tracing... Hit Ctrl-C to end.
^C
[...]

 CMD: tar              PID: 7491

           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       273
            >= 1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   6777
```

See Also

For more examples of DNLC tracing using DTrace, the DTraceToolkit has `dnlcstat` and `dnlcsnoop`, the latter printing DNLC lookup events as they occur; for example:

```
# dnlcsnoop.d
   PID CMD          TIME HIT PATH
  9185 bash            9   Y /etc
  9185 bash            3   Y /etc
 12293 bash            9   Y /usr
 12293 bash            3   Y /usr/bin
 12293 bash            4   Y /usr/bin/find
 12293 bash            7   Y /lib
 12293 bash            3   Y /lib/ld.so.1
 12293 find            6   Y /usr
 12293 find            3   Y /usr/bin
 12293 find            3   Y /usr/bin/find
[...]
```

fsflush_cpu.d

`fsflush` is the kernel file system flush thread on Oracle Solaris, which scans memory periodically for dirty data (data written to DRAM but not yet written to stable storage devices) and issues device writes to send it to disk. This thread applies to different file systems including UFS but does not apply to ZFS, which has its own way of flushing written data (transaction group sync).

Since system memory had become large (from megabytes to gigabytes since fsflush was written), the CPU time for `fsflush` to scan memory had become a performance issue that needed observability; at the time, DTrace didn't exist, and this was solved by adding a virtual process to `/proc` with the name `fsflush` that could be examined using standard process-monitoring tools (`ps(1)`, `prstat(1M)`):

```
solaris# ps -ecf | grep fsflush
    root     3      0  SYS  60   Nov 14 ?        1103:59 fsflush
```

Note the `SYS` scheduling class, identifying that this is a kernel thread.

The `fsflush_cpu.d` script prints `fsflush` information including the CPU time using DTrace.

Script

This script uses the fbt provider to trace the `fsflush_do_pages()` function and its logical calls to write data using `fop_putpage()`. The io provider is also used to measure physical device I/O triggered by `fsflush`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       trace("Tracing fsflush...\n");
```

continues

```
8           @fopbytes = sum(0); @iobytes = sum(0);
9   }
10
11  fbt::fsflush_do_pages:entry
12  {
13          self->vstart = vtimestamp;
14  }
15
16  fbt::fop_putpage:entry
17  /self->vstart/
18  {
19          @fopbytes = sum(arg2);
20  }
21
22  io:::start
23  /self->vstart/
24  {
25          @iobytes = sum(args[0]->b_bcount);
26          @ionum = count();
27  }
28
29  fbt::fsflush_do_pages:return
30  /self->vstart/
31  {
32          normalize(@fopbytes, 1024);
33          normalize(@iobytes, 1024);
34          this->delta = (vtimestamp - self->vstart) / 1000000;
35          printf("%Y %4d ms, ", walltimestamp, this->delta);
36          printa("fop: %7@d KB, ", @fopbytes);
37          printa("device: %7@d KB ", @iobytes);
38          printa("%5@d I/O", @ionum);
39          printf("\n");
40          self->vstart = 0;
41          clear(@fopbytes); clear(@iobytes); clear(@ionum);
42  }
```

Script fsflush_cpu.d

Script subtleties include the following.

- Lines 19, 25, and 26 use aggregations instead of global variables, for reliability on multi-CPU environments.
- Lines 36 to 38 print aggregations in separate `printa()` statements instead of a single statement, so this worked on the earliest versions of DTrace on Oracle Solaris, when support for multiple aggregations in a single `printa()` did not yet exist.
- Line 8 and using `clear()` instead of `trunc()` on line 41 are intended to ensure that the aggregations will be printed. Without them, if an aggregation contains no data, the `printa()` statement will be skipped, and the output line will miss elements.
- Since only `fsflush_do_pages()` is traced, only the flushing of pages is considered in the CPU time reported, not the flushing of inodes (the script could be enhanced to trace that as well).

Example

A line is printed for each `fsflush` run, showing the CPU time spent in `fsflush`, the amount of logical data written via the `fop` interface, and the number of physical data writes issued to the storage devices including the physical I/O count:

```
# fsflush_cpu.d
Tracing fsflush...
2010 Jun 20 04:15:52   24 ms, fop:     228 KB, device:     216 KB    54 I/O
2010 Jun 20 04:15:53   26 ms, fop:     260 KB, device:     244 KB    61 I/O
2010 Jun 20 04:15:54   35 ms, fop:    1052 KB, device:    1044 KB   261 I/O
2010 Jun 20 04:15:56   52 ms, fop:    1548 KB, device:    1532 KB   383 I/O
2010 Jun 20 04:15:57   60 ms, fop:    2756 KB, device:    2740 KB   685 I/O
2010 Jun 20 04:15:58   41 ms, fop:    1484 KB, device:    1480 KB   370 I/O
2010 Jun 20 04:15:59   37 ms, fop:    1284 KB, device:    1272 KB   318 I/O
2010 Jun 20 04:16:00   38 ms, fop:     644 KB, device:     632 KB   157 I/O
[...]
```

To demonstrate this, we needed dirty data for `fsflush` to write out. We did this by writing data to a UFS file system, performing a random 4KB write workload to a large file.

We found that applying a sequential write workload did not leave dirty data for `fsflush` to pick up, meaning that the writes to disk were occurring via a different code path. That different code path can be identified using DTrace, by looking at the stack backtraces when disk writes are being issued:

```
# dtrace -n 'io:::start /!(args[0]->b_flags & B_READ)/ { @[stack()] = count(); }'
dtrace: description 'io:::start ' matched 6 probes
^C
[...]
              ufs`lufs_write_strategy+0x100
              ufs`ufs_putapage+0x439
              ufs`ufs_putpages+0x308
              ufs`ufs_putpage+0x82
              genunix`fop_putpage+0x28
              genunix`segmap_release+0x24f
              ufs`wrip+0x4b5
              ufs`ufs_write+0x211
              genunix`fop_write+0x31
              genunix`write+0x287
              genunix`write32+0xe
              unix`sys_syscall32+0x101
             3201
```

So, `fop_putpage()` is happening directly from the `ufs_write()`, rather than `fsflush`.

fsflush.d

The previous script (`fsflush_cpu.d`) was an example of using DTrace to create statistics of interest. This is an example of retrieving existing kernel statistics—if they are available—and printing them out. It was written by Jon Haslam[4] and published in *Solaris Internals* (McDougall and Mauro, 2006).

Statistics are maintained in the kernel to count `fsflush` pages scanned, modified pages found, run time (CPU time), and more.

```
usr/src/uts/common/fs/fsflush.c:
    /*
     * some statistics for fsflush_do_pages
     */
    typedef struct {
            ulong_t fsf_scan;       /* number of pages scanned */
            ulong_t fsf_examined;   /* number of page_t's actually examined, can */
                                    /* be less than fsf_scan due to large pages */
            ulong_t fsf_locked;     /* pages we actually page_lock()ed */
            ulong_t fsf_modified;   /* number of modified pages found */
            ulong_t fsf_coalesce;   /* number of page coalesces done */
            ulong_t fsf_time;       /* nanoseconds of run time */
            ulong_t fsf_releases;   /* number of page_release() done */
    } fsf_stat_t;

    fsf_stat_t fsf_recent;  /* counts for most recent duty cycle */
    fsf_stat_t fsf_total;   /* total of counts */
```

They are kept in a global variable called `fsf_total` of `fsf_stat_t`, which the `fsflush.d` script reads using the ` kernel variable prefix.

Script

Since the counters are incremental, it prints out the delta every second:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

BEGIN
{
   lexam = 0; lscan = 0; llock = 0; lmod = 0; lcoal = 0; lrel = 0; lti = 0;
   printf("%10s %10s %10s %10s %10s %10s %10s\n", "SCANNED", "EXAMINED",
       "LOCKED", "MODIFIED", "COALESCE", "RELEASES", "TIME(ns)");
}

tick-1s
/lexam/
{
    printf("%10d %10d %10d %10d %10d %10d %10d\n", `fsf_total.fsf_scan,
        `fsf_total.fsf_examined - lexam, `fsf_total.fsf_locked - llock,
        `fsf_total.fsf_modified - lmod, `fsf_total.fsf_coalesce - lcoal,
        `fsf_total.fsf_releases - lrel, `fsf_total.fsf_time - ltime);
    lexam = `fsf_total.fsf_examined;
    lscan = `fsf_total.fsf_scan;
    llock = `fsf_total.fsf_locked;
    lmod = `fsf_total.fsf_modified;
    lcoal = `fsf_total.fsf_coalesce;
```

4. This was originally posted at *http://blogs.sun.com/jonh/entry/fsflush_revisited_in_d*.

```
    lrel = `fsf_total.fsf_releases;
    ltime = `fsf_total.fsf_time;
}

/*
 * First time through
 */

tick-1s
/!lexam/
{
    lexam = `fsf_total.fsf_examined;
    lscan = `fsf_total.fsf_scan;
    llock = `fsf_total.fsf_locked;
    lmod = `fsf_total.fsf_modified;
    lcoal = `fsf_total.fsf_coalesce;
    ltime = `fsf_total.fsf_time;
    lrel = `fsf_total.fsf_releases;
}
```

Script fsflush.d

This script uses the profile provider for the `tick-1s` probes, which is a stable provider. The script itself isn't considered stable, because it retrieves kernel internal statistics that may be subject to change (`fsf_stat_t`).

Example

```
solaris# fsflush.d
   SCANNED   EXAMINED     LOCKED   MODIFIED   COALESCE   RELEASES   TIME(ns)
     34871      34872       2243        365          0          0    3246343
     34871      34872       1576        204          0          0    2727493
     34871      34872       1689        221          0          0    2904566
     34871      34872        114         19          0          0    2221724
     34871      34872       1849        892          0          0    3297796
     34871      34872       1304        517          0          0    3408503
[...]
```

UFS Scripts

UFS is the Unix File System, based on Fast File System (FFS), and was the main file system used by Solaris until ZFS. UFS exists on other operating systems, including FreeBSD, where it can also be examined using DTrace. Although the on-disk structures and basic operation of UFS are similar, the implementation of UFS differs between operating systems. This is noticeable when listing the UFS probes via the fbt provider:

```
solaris# dtrace -ln 'fbt::ufs_*:' | wc -l
     403

freebsd# dtrace -ln 'fbt::ufs_*:' | wc -l
     107
```

For comparison, only those beginning with `ufs_` are listed. The fbt provider on Oracle Solaris can match the module name as `ufs`, so the complete list of UFS probes can be listed using `fbt:ufs::` (which shows 832 probes).

This section demonstrates UFS tracing on Oracle Solaris and is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. A basic understanding of UFS internals is assumed, which you can study in Chapter 15, The UFS File System, of *Solaris Internals* (McDougall and Mauro, 2006).

Since there is currently no stable UFS provider, the fbt[5] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for UFS analysis.

ufssnoop.d

This script uses the fbt provider to trace and print UFS calls from within the ufs kernel module. It provides a raw dump of what UFS is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for post-processing, such as filtering for events of interest.

The script is included here to show that this is possible and how it might look. This is written for a particular version of Oracle Solaris ZFS and will need tweaks to work on other versions. The functionality and output is similar to `solvfssnoop.d` shown earlier.

Script

Common UFS requests are traced: See the probe names on lines 33 to 35. This script can be enhanced to include more request types as desired: See the source file on line 12 for the list.

```
1    #!/usr/sbin/dtrace -Zs
2
3    #pragma D option quiet
4    #pragma D option switchrate=10hz
5
6    dtrace:::BEGIN
7    {
8            printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
9                "PID", "PROCESS", "CALL", "KB", "PATH");
10   }
```

5. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```
11
12  /* see uts/common/fs/ufs/ufs_vnops.c */
13
14  fbt::ufs_read:entry, fbt::ufs_write:entry
15  {
16          self->path = args[0]->v_path;
17          self->kb = args[1]->uio_resid / 1024;
18  }
19
20  fbt::ufs_open:entry
21  {
22          self->path = (*(struct vnode **)arg0)->v_path;
23          self->kb = 0;
24  }
25
26  fbt::ufs_close:entry, fbt::ufs_ioctl:entry, fbt::ufs_getattr:entry,
27  fbt::ufs_readdir:entry
28  {
29          self->path = args[0]->v_path;
30          self->kb = 0;
31  }
32
33  fbt::ufs_read:entry, fbt::ufs_write:entry, fbt::ufs_open:entry,
34  fbt::ufs_close:entry, fbt::ufs_ioctl:entry, fbt::ufs_getattr:entry,
35  fbt::ufs_readdir:entry
36  {
37          printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
38              uid, pid, execname, probefunc, self->kb,
39              self->path != NULL ? stringof(self->path) : "<null>");
40          self->path = 0; self->kb = 0;
41  }
```

Script ufssnoop.d

As another lesson in the instability of the fbt provider, the `ufs_open()` call doesn't exist on earlier versions of UFS. For this script to provide some functionality without it, the `-Z` option is used on line 1 so that the script will execute despite missing a probe, and line 22 casts `arg0` instead of using `args[0]` so that the script compiles.

Example

To test this script, the `dd(1)` command was used to perform three 8KB reads from a file:

```
solaris# ufssnoop.d
TIME(ms)        UID    PID PROCESS      CALL         KB   PATH
1155732900        0   8312 dd           ufs_open     0    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_read     8    /mnt/1m
1155732901        0   8312 dd           ufs_close    0    /mnt/1m
1155739611        0   8313 ls           ufs_getattr  0    /mnt
1155739611        0   8313 ls           ufs_getattr  0    /mnt
[...]
```

The events have been traced correctly. The `TIME(ms)` column showed no delay between these reads, suggesting that the data returned from DRAM cache. This column can also be used for postsorting, because the output may become shuffled slightly on multi-CPU systems.

ufsreadahead.d

Oracle Solaris UFS uses read-ahead to improve the performance of sequential workloads. This is where a sequential read pattern is detected, allowing UFS to predict the next requested reads and issue them before they are actually requested, to prewarm the cache.

The `ufsreadahead.d` script shows bytes read by UFS and those requested by read-ahead. This can be used on a known sequential workload to check that read-ahead is working correctly and also on an unknown workload to determine whether it is sequential or random.

Script

Since this script is tracing UFS internals using the fbt provider and will require maintenance, it has been kept as simple as possible:

```
#!/usr/sbin/dtrace -s

fbt::ufs_getpage:entry
{
        @["UFS read (bytes)"] = sum(arg2);
}

fbt::ufs_getpage_ra:return
{
        @["UFS read ahead (bytes)"] = sum(arg1);
}
```

Script ufsreadahead.d

Example

The following example shows the use of `ufsreadahead.d` examining a sequential/streaming read workload:

```
solaris# ufsreadahead.d
dtrace: script './ufsreadahead.d' matched 2 probes
^C

  UFS read ahead (bytes)                                    70512640
  UFS read (bytes)                                          71675904
```

This was a known sequential read workload. The output shows that about 71MB were reads from UFS and 70MB were from read-ahead, suggesting that UFS has correctly detected this as sequential. (It isn't certain, since the script isn't checking that the read-ahead data was then actually read by anyone.)

Here we see the same script applied to a random read workload:

```
solaris# ufsreadahead.d
dtrace: script './ufsreadahead.d' matched 2 probes
^C

  UFS read (bytes)                                            2099136
```

This was a known random read workload that performed 2MB of reads from UFS. No read-ahead was triggered, which is what we would expect (hope).

See Also

For more examples of UFS read-ahead analysis using DTrace, see the `fspaging.d` and `fsrw.d` scripts from the DTraceToolkit, which trace I/O from the syscall layer to the storage device layer. Here's an example:

```
solaris# fsrw.d
Event            Device RW     Size Offset Path
sc-read               .  R     8192      0 /mnt/bigfile
  fop_read            .  R     8192      0 /mnt/bigfile
    disk_io        sd15  R     8192      0 /mnt/bigfile
    disk_ra        sd15  R     8192      8 /mnt/bigfile
sc-read               .  R     8192      8 /mnt/bigfile
  fop_read            .  R     8192      8 /mnt/bigfile
    disk_ra        sd15  R    81920     16 /mnt/bigfile
    disk_ra        sd15  R     8192     96 <none>
    disk_ra        sd15  R     8192     96 /mnt/bigfile
sc-read               .  R     8192     16 /mnt/bigfile
  fop_read            .  R     8192     16 /mnt/bigfile
    disk_ra        sd15  R   131072    104 /mnt/bigfile
    disk_ra        sd15  R  1048576    232 /mnt/bigfile
sc-read               .  R     8192     24 /mnt/bigfile
  fop_read            .  R     8192     24 /mnt/bigfile
sc-read               .  R     8192     32 /mnt/bigfile
  fop_read            .  R     8192     32 /mnt/bigfile
[...]
```

This output shows five syscall reads (`sc-read`) of 8KB in size, starting from file offset 0 and reaching file offset 32 (kilobytes). The first of these syscall reads triggers an 8KB VFS read (`fop_read`), which triggers a disk read to satisfy it (`disk_io`); also at this point, UFS read-ahead triggers the next 8KB to be read from disk (`disk_ra`). The next syscall read triggers three more read-aheads. The last read-ahead seen in this output shows a 1MB read from offset 232, and yet the syscall interface—what's actually being requested of UFS—has only had three 8KB reads at this point. That's optimistic!

ufsimiss.d

The Oracle Solaris UFS implementation uses an inode cache to improve the performance of inode queries. There are various kernel statistics we can use to observe the performance of this cache, for example:

```
solaris# kstat -p ufs::inode_cache:hits ufs::inode_cache:misses 1
ufs:0:inode_cache:hits  580003
ufs:0:inode_cache:misses        1294907

ufs:0:inode_cache:hits  581810
ufs:0:inode_cache:misses        1299367

ufs:0:inode_cache:hits  582973
ufs:0:inode_cache:misses        1304608
[...]
```

These counters show a high rate of inode cache misses. DTrace can investigate these further: The `ufsimiss.d` script shows the process and filename for each inode cache miss.

Script

The parent directory vnode and filename pointers are cached on `ufs_lookup()` for later printing if an inode cache miss occurred, and `ufs_alloc_inode()` was entered:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
       printf("%6s %-16s %s\n", "PID", "PROCESS", "INODE MISS PATH");
}

fbt::ufs_lookup:entry
{
        self->dvp = args[0];
        self->name = arg1;
}

fbt::ufs_lookup:return
{
        self->dvp = 0;
        self->name = 0;
}

fbt::ufs_alloc_inode:entry
```

```
/self->dvp && self->name/
{
        printf("%6d %-16s %s/%s\n", pid, execname,
            stringof(self->dvp->v_path), stringof(self->name));
}
```

Script ufsimiss.d

Example

Here the UFS inode cache misses were caused by `find(1)` searching `/usr/share/man`:

```
solaris# ufsimiss.d
   PID PROCESS          INODE MISS PATH
 22966 find             /usr/share/man/sman3tiff/TIFFCheckTile.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFClientOpen.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFCurrentRow.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFDefaultStripSize.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFFileno.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFGetVersion.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsMSB2LSB.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsTiled.3tiff
 22966 find             /usr/share/man/sman3tiff/TIFFIsUpSampled.3tiff
[...]
```

ZFS Scripts

ZFS is an advanced file system and volume manager available on Oracle Solaris. Its features include 128-bit capacity, different RAID types, copy-on-write transactions, snapshots, clones, dynamic striping, variable block size, end-to-end checksumming, built-in compression, data-deduplication, support for hybrid storage pools, quotas, and more. The interaction of these features is interesting for those examining file system performance, and they have become a common target for DTrace.

ZFS employs an I/O pipeline (ZIO) that ends with aggregation of I/O at the device level. By the time an I/O is sent to disk, the content may refer to multiple files (specifically, there is no longer a single `vnode_t` for that I/O). Because of this, the io provider on ZFS can't show the path name for I/O; this has been filed as a bug (CR 6266202 "DTrace io provider doesn't work with ZFS"). At the time of writing, this bug has not been fixed. The ZFS path name of disk I/O can still be fetched with a little more effort using DTrace; the `ziosnoop.d` script described next shows one way to do this. For reads, it may be possible to simply identify slow reads at the ZFS interface, as demonstrated by the `zfsslower.d` script.

This section demonstrates ZFS tracing on Oracle Solaris and is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. An understanding of ZFS internals is assumed.

Since there is currently no stable ZFS provider, the fbt[6] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for ZFS analysis.

zfssnoop.d

This script uses the fbt provider to trace and print ZFS calls from within the zfs kernel module. It provides a raw dump of what ZFS is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for postprocessing, such as filtering for events of interest. The functionality and output is similar to `solvfssnoop.d` shown earlier.

Script

Common ZFS requests are traced; see the probe names on lines 33 to 35. This script can be enhanced to include more request types as desired; see the source file on line 12 for the list.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           printf("%-12s %6s %6s %-12.12s %-12s %-4s %s\n", "TIME(ms)", "UID",
9               "PID", "PROCESS", "CALL", "KB", "PATH");
10  }
11
12  /* see uts/common/fs/zfs/zfs_vnops.c */
13
14  fbt::zfs_read:entry, fbt::zfs_write:entry
15  {
16          self->path = args[0]->v_path;
17          self->kb = args[1]->uio_resid / 1024;
18  }
19
20  fbt::zfs_open:entry
21  {
22          self->path = (*args[0])->v_path;
23          self->kb = 0;
24  }
25
26  fbt::zfs_close:entry, fbt::zfs_ioctl:entry, fbt::zfs_getattr:entry,
27  fbt::zfs_readdir:entry
```

6. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```
{
        self->path = args[0]->v_path;
        self->kb = 0;
}

fbt::zfs_read:entry, fbt::zfs_write:entry, fbt::zfs_open:entry,
fbt::zfs_close:entry, fbt::zfs_ioctl:entry, fbt::zfs_getattr:entry,
fbt::zfs_readdir:entry
{
        printf("%-12d %6d %6d %-12.12s %-12s %-4d %s\n", timestamp / 1000000,
            uid, pid, execname, probefunc, self->kb,
            self->path != NULL ? stringof(self->path) : "<null>");
        self->path = 0; self->kb = 0;
}
```

Script zfssnoop.d

The TIME(ms) column can be used for postsorting, because the output may become shuffled slightly on multi-CPU systems.

Example

The following script was run on a desktop to identify ZFS activity:

```
solaris# zfssnoop.d
TIME(ms)         UID    PID PROCESS      CALL         KB   PATH
19202174470      102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470      102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470      102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470      102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.gnome2/
vfolders
19202174470      102  19981 gnome-panel  zfs_getattr  0    /export/home/claire/.recentl
y-used
19202175400      101   2903 squid        zfs_open     0    /squidcache/05/03
19202175400      101   2903 squid        zfs_getattr  0    /squidcache/05/03
19202175400      101   2903 squid        zfs_readdir  0    /squidcache/05/03
19202175400      101   2903 squid        zfs_readdir  0    /squidcache/05/03
19202175400      101   2903 squid        zfs_close    0    /squidcache/05/03
19202175427      102  23885 firefox-bin  zfs_getattr  0    /export/home/claire/.recentl
yused.xbe
l
19202176030      102  13622 nautilus     zfs_getattr  0    /export/home/claire/Desktop
19202176215      102  23885 firefox-bin  zfs_read     3    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_002_
19202176216      102  23885 firefox-bin  zfs_read     3    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_002_
19202176215      102  23885 firefox-bin  zfs_read     0    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_001_
19202176216      102  23885 firefox-bin  zfs_read     0    /export/home/claire/.mozilla
/firefox/3c8k4kh0.default/Cache/_CACHE_001_
[...]
```

Various ZFS calls have been traced, including gnome-panel checking file attributes and firefox-bin reading cache files.

zfsslower.d

This is a variation of the `zfssnoop.d` script intended for the analysis of performance issues. `zfsslower.d` shows the time for read and write I/O in milliseconds. A minimum number of milliseconds can be provided as an argument when running the script, which causes it to print only I/O equal to or slower than the provided milliseconds.

Because of CR 6266202 (mentioned earlier), we currently cannot trace disk I/O with ZFS filename information using the io provider arguments. `zfsslower.d` may be used as a workaround: By executing it with a minimum time that is likely to ensure that it is disk I/O (for example, at least 2 ms), we can trace likely disk I/O events with ZFS filename information.

Script

The `defaultargs` pragma is used on line 4 so that an optional argument can be provided of the minimum I/O time to print. If no argument is provided, the minimum time is zero, since `$1` will be 0 on line 11.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option defaultargs
5   #pragma D option switchrate=10hz
6
7   dtrace:::BEGIN
8   {
9           printf("%-20s %-16s %1s %4s %6s %s\n", "TIME", "PROCESS",
10              "D", "KB", "ms", "FILE");
11          min_ns = $1 * 1000000;
12  }
13
14  /* see uts/common/fs/zfs/zfs_vnops.c */
15
16  fbt::zfs_read:entry, fbt::zfs_write:entry
17  {
18          self->path = args[0]->v_path;
19          self->kb = args[1]->uio_resid / 1024;
20          self->start = timestamp;
21  }
22
23  fbt::zfs_read:return, fbt::zfs_write:return
24  /self->start && (timestamp - self->start) >= min_ns/
25  {
26          this->iotime = (timestamp - self->start) / 1000000;
27          this->dir = probefunc == "zfs_read" ? "R" : "W";
28          printf("%-20Y %-16s %1s %4d %6d %s\n", walltimestamp,
29              execname, this->dir, self->kb, this->iotime,
30              self->path != NULL ? stringof(self->path) : "<null>");
31  }
32
33  fbt::zfs_read:return, fbt::zfs_write:return
34  {
35          self->path = 0; self->kb = 0; self->start = 0;
36  }
```

Script zfsslower.d

Example

Here the `zfsslower.d` script was run with an argument of 1 to show only ZFS reads and writes that took 1 millisecond or longer:

```
solaris# zfsslower.d 1
TIME                    PROCESS          D   KB      ms FILE
2010 Jun 26 03:28:49 cat              R    8      14 /export/home/brendan/randread.pl
2010 Jun 26 03:29:04 cksum            R    4       5 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      20 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      34 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4       7 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      12 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4       1 /export/home/brendan/perf.tar
2010 Jun 26 03:29:04 cksum            R    4      81 /export/home/brendan/perf.tar
[...]
```

The files accessed here were not cached and had to be read from disk.

zioprint.d

The ZFS I/O pipeline (ZIO) is of particular interest for performance analysis or troubleshooting, because it processes, schedules, and issues device I/O. It does this through various stages whose function names (and hence fbt provider probe names) have changed over time. Because of this, a script that traces specific ZIO functions would execute only on a particular kernel version and would require regular maintenance to match kernel updates.

The `zioprint.d` script addresses this by matching all zio functions using a wildcard, dumping data generically, and leaving the rest to postprocessing of the output (for example, using Perl).

Script

This script prints the first five arguments on function entry as hexadecimal integers, whether or not that's meaningful (which can be determined later during postprocessing). For many of these functions, the first argument on entry is the address of a `zio_t`, so a postprocessor can use that address as a key to follow that zio through the stages. The return offset and value are also printed.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-3s %-22s %-6s %s\n", "TIME(us)", "CPU", "FUNC",
            "NAME", "ARGS");
}
```

continues

```
11
12  fbt::zio_*:entry
13  {
14          printf("%-16d %-3d %-22s %-6s %x %x %x %x %x\n", timestamp / 1000,
15              cpu, probefunc, probename, arg0, arg1, arg2, arg3, arg4);
16  }
17
18  fbt::zio_*:return
19  {
20          printf("%-16d %-3d %-22s %-6s %x %x\n", timestamp / 1000, cpu,
21              probefunc, probename, arg0, arg1);
22  }
```

Script zioprint.d

This script can be reused to dump events from any kernel area by changing the probe names on lines 12 and 18.

Example

The script is intended to be used to write a dump file (either by using shell redirection > or via the `dtrace(1M) -o` option) for postprocessing. Since the script is generic, it is likely to execute on any kernel version and produce a dump file, which can be especially handy in situations with limited access to the target system but unlimited access to any other system (desktop/laptop) for postprocessing.

```
solaris# zioprint.d
TIME(us)         CPU FUNC                   NAME   ARGS
1484927856573    0   zio_taskq_dispatch     entry  ffffff4136711c98 2 0 4a 49
1484927856594    0   zio_taskq_dispatch     return ac ffffff4456fc8090
1484927856616    0   zio_interrupt          return 1d ffffff4456fc8090
1484927856630    0   zio_execute            entry  ffffff4136711c98 ffffff4456fc8090
a477aa00 a477aa00 c2244e36f410a
1484927856643    0   zio_vdev_io_done       entry  ffffff4136711c98 ffffff4456fc8090
a477aa00 a477aa00 12
1484927856653    0   zio_wait_for_children  entry  ffffff4136711c98 0 1 a477aa00 12
1484927856658    0   zio_wait_for_children  return 7b 0
1484927856667    0   zio_vdev_io_done       return 117 100
[...]
```

The meaning of each hexadecimal argument can be determined by reading the ZFS source for that kernel version. For example, the `zio_wait_for_children()` calls shown earlier have the function prototype:

```
usr/src/uts/common/fs/zfs/zio.c:

static boolean_t
zio_wait_for_children(zio_t *zio, enum zio_child child, enum zio_wait_type wait)
```

which means that the entry traced earlier has a `zio_t` with address `ffffff4136711c98` and a `zio_wait_type` of 1 (`ZIO_WAIT_DONE`). The additional arguments printed (`a477aa00` and `12`) are leftover register values that are not part of the function entry arguments.

ziosnoop.d

The `ziosnoop.d` script is an enhancement of `zioprint.d`, by taking a couple of the functions and printing useful information from the kernel—including the pool name and file path name. The trade-off is that these additions make the script more fragile and may require maintenance to match kernel changes.

Script

The `zio_create()` and `zio_done()` functions were chosen as start and end points for ZIO (`zio_destroy()` may be a better endpoint, but it didn't exist on earlier kernel versions). For `zio_create()`, information about the requested I/O including pool name and file path name (if known) are printed. On `zio_done()`, the results of the I/O, including device path (if present) and error values, are printed.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        start = timestamp;
        printf("%-10s %-3s %-12s %-16s %s\n", "TIME(us)", "CPU",
            "ZIO_EVENT", "ARG0", "INFO (see script)");
}

fbt::zfs_read:entry, fbt::zfs_write:entry   { self->vp = args[0]; }
fbt::zfs_read:return, fbt::zfs_write:return { self->vp = 0; }

fbt::zio_create:return
/$1 || args[1]->io_type/
{
        /* INFO: pool zio_type zio_flag bytes path */
        printf("%-10d %-3d %-12s %-16x %s %d %x %d %s\n",
            (timestamp - start) / 1000, cpu, "CREATED", arg1,
            stringof(args[1]->io_spa->spa_name), args[1]->io_type,
            args[1]->io_flags, args[1]->io_size, self->vp &&
            self->vp->v_path ? stringof(self->vp->v_path) : "<null>");
}

fbt::zio_*:entry
/$1/
{
        printf("%-10d %-3d %-12s %-16x\n", (timestamp - start) / 1000, cpu,
            probefunc, arg0);
}
```

continues

```

fbt::zio_done:entry
/$1 || args[0]->io_type/
{
        /* INFO: io_error vdev_state vdev_path */
        printf("%-10d %-3d %-12s %-16x %d %d %s\n", (timestamp - start) / 1000,
            cpu, "DONE", arg0, args[0]->io_error,
            args[0]->io_vd ? args[0]->io_vd->vdev_state : 0,
            args[0]->io_vd && args[0]->io_vd->vdev_path ?
            stringof(args[0]->io_vd->vdev_path) : "<null>");
}
```

Script ziosnoop.d

By default, only `zio_create()` and `zio_done()` are traced; if an optional argument of 1 (nonzero) is provided, the script traces all other zio functions as well.

Examples

This is the default output:

```
solaris# ziosnoop.d
TIME(us)   CPU ZIO_EVENT    ARG0             INFO (see script)
75467      2   CREATED      ffffff4468f79330 pool0 1 40440 131072 /pool0/fs1/1t
96330      2   CREATED      ffffff44571b1360 pool0 1 40 131072 /pool0/fs1/1t
96352      2   CREATED      ffffff46510a7cc0 pool0 1 40440 131072 /pool0/fs1/1t
96363      2   CREATED      ffffff4660b4a048 pool0 1 40440 131072 /pool0/fs1/1t
24516      5   DONE         ffffff59a619ecb0 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
24562      5   DONE         ffffff4141ecd340 0 7 <null>
24578      5   DONE         ffffff4465456320 0 0 <null>
34836      5   DONE         ffffff4141f8dca8 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
34854      5   DONE         ffffff414d8e8368 0 7 <null>
34867      5   DONE         ffffff446c3de9b8 0 0 <null>
44818      5   DONE         ffffff5b3defd968 0 7 /dev/dsk/c0t5000CCA20ED60164d0s0
[...]
```

Note the `TIME(us)` column—the output is shuffled. To see it in the correct order, write to a file and postsort on that column.

Running `ziosnoop.d` with an argument of 1 will execute verbose mode, printing all zio calls. Here it is written to a file, from which a particular `zio_t` address is searched using `grep(1)`:

```
solaris# ziosnoop.d 1 -o ziodump
solaris# more ziodump
TIME(us)   CPU ZIO_EVENT    ARG0             INFO (see script)
[...]
171324     6   CREATED      ffffff6440130368 pool0 1 40440 131072 /pool0/fs1/1t
171330     6   zio_nowait   ffffff6440130368
171332     6   zio_execute  ffffff6440130368
[...]
solaris# grep ffffff6440130368 ziodump | sort -n +0
```

```
171324     6   CREATED      ffffff6440130368 pool0 1 40440 131072 /pool0/fs1/1t
171330     6   zio_nowait   ffffff6440130368
171332     6   zio_execute  ffffff6440130368
171334     6   zio_vdev_io_start ffffff6440130368
179672     0   zio_interrupt ffffff6440130368
179676     0   zio_taskq_dispatch ffffff6440130368
179689     0   zio_execute  ffffff6440130368
179693     0   zio_vdev_io_done ffffff6440130368
179695     0   zio_wait_for_children ffffff6440130368
179698     0   zio_vdev_io_assess ffffff6440130368
179700     0   zio_wait_for_children ffffff6440130368
179702     0   zio_checksum_verify ffffff6440130368
179705     0   zio_checksum_error ffffff6440130368
179772     0   zio_done     ffffff6440130368
179775     0   DONE         ffffff6440130368 0 7 /dev/dsk/c0t5000CCA20ED60516d0s0
[...]
```

The output of grep(1) is passed to sort(1) to print the events in the correct timestamp order. Here, all events from zio_create() to zio_done() can be seen, along with the time stamp. Note the jump in time between zio_vdev_io_start() and zio_interrupt() (171334 us to 179672 us = 8 ms)—this is the device I/O time. Latency in other zio stages can be identified in the same way (which can be expedited by writing a postprocessor).

ziotype.d

The ziotype.d script shows what types of ZIO are being created, printing a count every five seconds.

Script

A translation table for zio_type is included in the BEGIN action, based on zfs.h. If zfs.h changes with kernel updates, this table will need to be modified to match.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /* see /usr/include/sys/fs/zfs.h */
        ziotype[0] = "null";
        ziotype[1] = "read";
        ziotype[2] = "write";
        ziotype[3] = "free";
        ziotype[4] = "claim";
        ziotype[5] = "ioctl";
        trace("Tracing ZIO...  Output interval 5 seconds, or Ctrl-C.\n");
}

fbt::zio_create:return
/args[1]->io_type/              /* skip null */
{
        @[stringof(args[1]->io_spa->spa_name),
            ziotype[args[1]->io_type] != NULL ?
```

continues

```
                ziotype[args[1]->io_type] : "?"] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n %-32s %-10s %10s\n", "POOL", "ZIO_TYPE", "CREATED");
        printa(" %-32s %-10s %@10d\n", @);
        trunc(@);
}
```

Script zioype.d

Example

The example has identified a mostly write workload of about 12,000 write ZIO every five seconds:

```
solaris# ziotype.d
Tracing ZIO...  Output interval 5 seconds, or Ctrl-C.

 POOL                             ZIO_TYPE      CREATED
 pool0                            ioctl              28
 pool0                            free               48
 pool0                            read             1546
 pool0                            write           12375

 POOL                             ZIO_TYPE      CREATED
 pool0                            ioctl              14
 pool0                            free               24
 pool0                            read             1260
 pool0                            write           11929
[...]
```

perturbation.d

The `perturbation.d` script measures ZFS read/write performance during a given perturbation. This can be used to quantify the performance impact during events such as snapshot creation.

Script

The perturbation function name is provided as an argument, which DTrace makes available in the script as `$$1`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs

dtrace:::BEGIN
{
        printf("Tracing ZFS perturbation by %s()... Ctrl-C to end.\n", $$1);
}
```

```

fbt::$$1:entry
{
        self->pstart = timestamp;
        perturbation = 1;
}

fbt::$$1:return
/self->pstart/
{
        this->ptime = (timestamp - self->pstart) / 1000000;
        @[probefunc, "perturbation duration (ms)"] = quantize(this->ptime);
        perturbation = 0;
}

fbt::zfs_read:entry, fbt::zfs_write:entry
{
        self->start = timestamp;
}

fbt::zfs_read:return, fbt::zfs_write:return
/self->start/
{
        this->iotime = (timestamp - self->start) / 1000000;
        @[probefunc, perturbation ? "during perturbation (ms)" :
            "normal (ms)"] = quantize(this->iotime);
        self->start = 0;
}
```

Script perturbation.d

Example

Here we measure ZFS performance during snapshot creation. The `perturbation.d` script is run with the argument `zfs_ioc_snapshot`, a function call that encompasses snapshot creation (for this kernel version). While tracing, a read and write workload was executing on ZFS, and three snapshots were created:

```
solaris# perturbation.d zfs_ioc_snapshot
Tracing ZFS perturbation by zfs_ioc_snapshot()... Ctrl-C to end.
^C

  zfs_write                                           normal (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 348381
               1 |                                         7
               2 |                                         0

  zfs_write                                           during perturbation (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 276029
               1 |                                         11
               2 |                                         5
               4 |                                         0

  zfs_ioc_snapshot                                    perturbation duration (ms)
           value  ------------- Distribution ------------- count
```

continues

```
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              2
            2048 |@@@@@@@@@@@@@                            1
            4096 |                                         0

  zfs_read                                         during perturbation (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@                                        5
               1 |                                         0
               2 |                                         0
               4 |                                         3
               8 |@@@@@@@@@@@@                             77
              16 |@@@@@@@@@@@@@@@@@@                       117
              32 |@@@@                                     26
              64 |@@                                       16
             128 |@                                        8
             256 |                                         2
             512 |@                                        5
            1024 |                                         0

  zfs_read                                         normal (ms)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@                                     97
               1 |                                         0
               2 |                                         0
               4 |@                                        29
               8 |@@@@@@@@@@@@@@@@@@@@@@@@                 563
              16 |@@@@@@@@@@                               241
              32 |                                         10
              64 |                                         1
             128 |                                         0
```

The impact on performance can be seen clearly in the last distribution plots for ZFS reads. In normal operation, the time for ZFS reads was mostly between 8 ms and 31 ms. During snapshot create, some ZFS reads were taking 32 ms and longer, with the slowest five I/O in the 512-ms to 1023-ms range. Fortunately, these are outliers: Most of the I/O was still in the 8-ms to 31-ms range, despite a snapshot being created.

Another target for `perturbation.d` can be the `spa_sync()` function.

Note that `perturbation.d` cannot be run without any arguments; if that is tried, DTrace will error because the `$$1` macro variable is undefined:

```
solaris# perturbation.d
dtrace: failed to compile script perturbation.d: line 11: invalid probe description "f
bt::$$1:entry": Undefined macro variable in probe description
```

A function name must be provided for DTrace to trace.

spasync.d

The `spa_sync()` function flushes a ZFS transaction group (TXG) to disk, which consists of dirty data written since the last `spa_sync()`.

Script

This script has a long history: Earlier versions were created by the ZFS engineering team and can be found in blog entries.[7] Here it has been rewritten to keep it short and to print only `spa_sync()` events that were longer than one millisecond—tunable on line 5:

```
1     #!/usr/sbin/dtrace -s
2
3     #pragma D option quiet
4
5     inline int MIN_MS = 1;
6
7     dtrace:::BEGIN
8     {
9           printf("Tracing ZFS spa_sync() slower than %d ms...\n", MIN_MS);
10          @bytes = sum(0);
11    }
12
13    fbt::spa_sync:entry
14    /!self->start/
15    {
16          in_spa_sync = 1;
17          self->start = timestamp;
18          self->spa = args[0];
19    }
20
21    io:::start
22    /in_spa_sync/
23    {
24          @io = count();
25          @bytes = sum(args[0]->b_bcount);
26    }
27
28    fbt::spa_sync:return
29    /self->start && (this->ms = (timestamp - self->start) / 1000000) > MIN_MS/
30    {
31          normalize(@bytes, 1048576);
32          printf("%-20Y %-10s %6d ms, ", walltimestamp,
33              stringof(self->spa->spa_name), this->ms);
34          printa("%@d MB %@d I/O\n", @bytes, @io);
35    }
36
37    fbt::spa_sync:return
38    {
39          self->start = 0; self->spa = 0; in_spa_sync = 0;
40          clear(@bytes); clear(@io);
41    }
```

Script spasync.d

7. See *http://blogs.sun.com/roch/entry/128k_suffice* by Roch Bourbonnais, and see *www.cuddletech.com/blog/pivot/entry.php?id=1015* by Ben Rockwood.

Example

```
solaris# spa_sync.d
Tracing ZFS spa_sync() slower than 1 ms...
2010 Jun 17 01:46:18 pool-0       2679 ms, 31 MB 2702 I/O
2010 Jun 17 01:46:18 pool-0        269 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 pool-0        108 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 system        597 ms, 0 MB 0 I/O
2010 Jun 17 01:46:18 pool-0        184 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 pool-0        154 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system        277 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system         34 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 pool-0        226 ms, 27 MB 1668 I/O
2010 Jun 17 01:46:19 system        262 ms, 0 MB 0 I/O
2010 Jun 17 01:46:19 system        174 ms, 0 MB 0 I/O
[...]
```

HFS+ Scripts

HFS+ is the Hierarchal File System plus from Apple, described in Technical Note TN1150[8] and *Mac OS X Internals*.

```
macosx# dtrace -ln 'fbt::hfs_*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 9396        fbt       mach_kernel                  hfs_addconverter entry
 9398        fbt       mach_kernel                          hfs_bmap entry
[...]
 9470        fbt       mach_kernel                    hfs_vnop_ioctl entry
 9472        fbt       mach_kernel           hfs_vnop_makenamedstream entry
 9474        fbt       mach_kernel                 hfs_vnop_offtoblk entry
 9476        fbt       mach_kernel                   hfs_vnop_pagein entry
 9478        fbt       mach_kernel                  hfs_vnop_pageout entry
 9480        fbt       mach_kernel                     hfs_vnop_read entry
 9482        fbt       mach_kernel         hfs_vnop_removenamedstream entry
 9484        fbt       mach_kernel                   hfs_vnop_select entry
 9486        fbt       mach_kernel                 hfs_vnop_strategy entry
 9488        fbt       mach_kernel                    hfs_vnop_write entry
```

Some of the functions in the HFS code are declared static, so their symbol information is not available for DTrace to probe. This includes `hfs_vnop_open()` and `hfs_vnop_close()`, which are missing from the previous list. Despite this, there are still enough visible functions from HFS+ for DTrace scripting: the functions that call HFS and the functions that HFS calls.

This section is intended for those wanting to dig deeper into file system internals, beyond what is possible at the syscall and VFS layers. A basic understanding of HFS+ internals is assumed, which can be studied in Chapter 12 of *Mac OS X Internals*.

8. See *http://developer.apple.com/mac/library/technotes/tn/tn1150.html*.

Since there is currently no stable HFS+ provider, the fbt[9] provider is used. fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. The following scripts were based on Mac OS X version 10.6 and may not work on other releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for HFS+ analysis.

hfssnoop.d

This script uses the fbt provider to trace HFS+ calls from within the kernel (this will need tweaks to work on future Mac OS X kernels). It provides a raw dump of what HFS+ is being requested to do, which can be useful for identifying load issues. Since the output is verbose and inclusive, it is suitable for postprocessing, such as filtering for events of interest. The functionality and output is similar to `macvfssnoop.d` shown earlier.

Script

This script currently only traces reads and writes. Other available `hfs_vnop_*` functions can be added, and those not visible (such as open) can be traced from an upper layer, such as VFS (via `VNOP_*`, and filtering on HFS calls only).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
      printf("%-12s %6s %6s %-12.12s %-14s %-4s %s\n", "TIME(ms)", "UID",
          "PID", "PROCESS", "CALL", "KB", "FILE");
}

/* see bsd/hfs/hfs_vnops.c */

fbt::hfs_vnop_read:entry
{
      this->read = (struct vnop_read_args *)arg0;
      self->path = this->read->a_vp->v_name;
      self->kb = this->read->a_uio->uio_resid_64 / 1024;
}

fbt::hfs_vnop_write:entry
{
      this->write = (struct vnop_write_args *)arg0;
      self->path = this->write->a_vp->v_name;
      self->kb = this->write->a_uio->uio_resid_64 / 1024;
}
```

continues

9. See the “fbt Provider” section in Chapter 12 for more discussion about use of the fbt provider.

```

fbt::hfs_vnop_read:entry, fbt::hfs_vnop_write:entry
{
      printf("%-12d %6d %6d %-12.12s %-14s %-4d %s\n", timestamp / 1000000,
          uid, pid, execname, probefunc, self->kb,
          self->path != NULL ? stringof(self->path) : "<null>");
      self->path = 0; self->kb = 0;
}
```

Script hfssnoop.d

Example

Here the `hfssnoop.d` script has traced `vim(1)` opening itself in another window to edit it:

```
macosx# hfssnoop.d
TIME(ms)         UID    PID PROCESS      CALL           KB   FILE
1311625280       501  67349 vim          hfs_vnop_read  4    LC_COLLATE
1311625280       501  67349 vim          hfs_vnop_read  0    LC_CTYPE/..namedfork/rsrc
1311625280       501  67349 vim          hfs_vnop_read  4    LC_CTYPE
[...]
1311625288       501  67349 vim          hfs_vnop_read  8    hfssnoop.d
1311625280       501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280       501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280       501  67349 vim          hfs_vnop_read  4    LC_CTYPE
1311625280       501  67349 vim          hfs_vnop_read  54   LC_CTYPE
1311625280       501  67349 vim          hfs_vnop_read  0    LC_MONETARY
1311625280       501  67349 vim          hfs_vnop_read  0    LC_NUMERIC
1311625280       501  67349 vim          hfs_vnop_read  0    LC_TIME
1311625280       501  67349 vim          hfs_vnop_read  0    LC_MESSAGES
1311625281       501  67349 vim          hfs_vnop_read  4    xterm-color
1311625282       501  67349 vim          hfs_vnop_read  4    vimrc
1311625282       501  67349 vim          hfs_vnop_read  4    vimrc
1311625284       501  67349 vim          hfs_vnop_read  4    netrwPlugin.vim
1311625284       501  67349 vim          hfs_vnop_read  4    netrwPlugin.vim
[...]
1311625285       501  67349 vim          hfs_vnop_read  4    zipPlugin.vim
1311625286       501  67349 vim          hfs_vnop_read  4    zipPlugin.vim
1311625288       501  67349 vim          hfs_vnop_write 4    .hfssnoop.d.swp
1311625288       501  67349 vim          hfs_vnop_read  64   hfssnoop.d
```

All the files read and written while vim was loading have been traced. The final lines show a swap file being written and vim reloading the `hfssnoop.d` file. The kilobyte sizes shown are those requested; many of these reads will have returned a smaller size in bytes (which can be shown, if desired, with more DTrace).

hfsslower.d

This is a variation of the `hfssnoop.d` script, intended for the analysis of performance issues. `hfsslower.d` shows the time for read and write I/O in milliseconds. A minimum number of milliseconds can be provided as an argument when running the script, which causes it to print only that I/O equal to or slower than the provided milliseconds.

Script

The `defaultargs` pragma is used on line 4 so that an optional argument can be provided of the minimum I/O time to print. If no argument is provided, the minimum time is zero, since `$1` will be 0 on line 11.

```
1     #!/usr/sbin/dtrace -s
2
3     #pragma D option quiet
4     #pragma D option defaultargs
5     #pragma D option switchrate=10hz
6
7     dtrace:::BEGIN
8     {
9           printf("%-20s %-16s %1s %4s %6s %s\n", "TIME", "PROCESS",
10              "D", "KB", "ms", "FILE");
11          min_ns = $1 * 1000000;
12    }
13
14    /* see bsd/hfs/hfs_vnops.c */
15
16    fbt::hfs_vnop_read:entry
17    {
18          this->read = (struct vnop_read_args *)arg0;
19          self->path = this->read->a_vp->v_name;
20          self->kb = this->read->a_uio->uio_resid_64 / 1024;
21          self->start = timestamp;
22    }
23
24    fbt::hfs_vnop_write:entry
25    {
26          this->write = (struct vnop_write_args *)arg0;
27          self->path = this->write->a_vp->v_name;
28          self->kb = this->write->a_uio->uio_resid_64 / 1024;
29          self->start = timestamp;
30    }
31
32    fbt::hfs_vnop_read:return, fbt::hfs_vnop_write:return
33    /self->start && (timestamp - self->start) >= min_ns/
34    {
35          this->iotime = (timestamp - self->start) / 1000000;
36          this->dir = probefunc == "hfs_vnop_read" ? "R" : "W";
37          printf("%-20Y %-16s %1s %4d %6d %s\n", walltimestamp,
38              execname, this->dir, self->kb, this->iotime,
39              self->path != NULL ? stringof(self->path) : "<null>");
40    }
41
42    fbt::hfs_vnop_read:return, fbt::hfs_vnop_write:return
43    {
44          self->path = 0; self->kb = 0; self->start = 0;
45    }
```

Script hfslower.d

Example

Here `hfsslower.d` is run with the argument 1 so that it prints out only the I/O that took one millisecond and longer:

```
macosx# hfsslower.d 1
TIME                 PROCESS          D   KB      ms FILE
2010 Jun 23 00:44:05 mdworker32       R    0      21 sandbox-cache.db
2010 Jun 23 00:44:05 mdworker32       R    0      19 AdiumSpotlightImporter
2010 Jun 23 00:44:05 mdworker32       R   16      18 schema.xml
2010 Jun 23 00:44:05 soffice          W    1       2 sve4a.tmp
2010 Jun 23 00:44:05 soffice          W    1       3 sve4a.tmp
2010 Jun 23 00:44:05 soffice          R   31       2 sve4a.tmp
2010 Jun 23 00:44:05 fontd            R    0      22 Silom.ttf/..namedfork/rsrc
^C
```

While tracing, there was many fast (less than 1 ms) I/Os to HFS that were filtered from the output.

hfsfileread.d

This script shows both logical (VFS) and physical (disk) reads to HFS+ files, showing data requests from the in-memory cache vs. disk.

Script

This script traces the size of read requests. The size of the returned data may be smaller than was requested or zero if the read failed; the returned size could also be traced if desired.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      trace("Tracing HFS+ file reads... Hit Ctrl-C to end.\n");
}

fbt::hfs_vnop_read:entry
{
      this->read = (struct vnop_read_args *)arg0;
      this->path = this->read->a_vp->v_name;
      this->bytes = this->read->a_uio->uio_resid_64;
      @r[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::hfs_vnop_strategy:entry
/((struct vnop_strategy_args *)arg0)->a_bp->b_flags & B_READ/
{
      this->strategy = (struct vnop_strategy_args *)arg0;
      this->path = this->strategy->a_bp->b_vp->v_name;
      this->bytes = this->strategy->a_bp->b_bcount;
      @s[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

dtrace:::END
{
      printf(" %-56s %10s %10s\n", "FILE", "READ(B)", "DISK(B)");
      printa(" %-56s %@10d %@10d\n", @r, @s);
}
```

Script hfsfileread.d

Example

While tracing, there were about 240MB of requested reads to the `ss7000_b00.vmdk` file, about 230MB of which were from disk, meaning that this file is mostly uncached. The `10m_file` was completely read; however, 0 bytes were read from disk, meaning that it was entirely cached.

```
macosx# hfsfileread.d
Tracing HFS+ file reads... Hit Ctrl-C to end.
^C
 FILE                                                        READ(B)    DISK(B)
 swapfile1                                                         0       4096
 dyld/..namedfork/rsrc                                            50          0
 dyld                                                           4636          0
 cksum                                                         12288          0
 template.odt                                                 141312     143360
 10m_file                                                   10502144          0
 ss7000_b00.vmdk                                           246251520  230264832
```

PCFS Scripts

pcfs is an Oracle Solaris driver for the Microsoft FAT16 and FAT32 file systems. Though it was once popular for diskettes, today FAT file systems are more likely to be found on USB storage devices.

Since there is currently no stable PCFS provider, the fbt provider is used here. fbt instruments a particular operating system and version, so this script may therefore require modifications to match the software version you are using.

pcfsrw.d

This script shows `read()`, `write()`, and `readdir()` calls to pcfs, with details including file path name and latency for the I/O in milliseconds.

Script

This script traces `pcfs` kernel functions; if the `pcfs` module is not loaded (no `pcfs` in use), the script will not execute because the functions will not yet be present in the kernel for DTrace to find and probe. If desired, the `-Z` option can be added to line 1, which would allow the script to be executed before `pcfs` was loaded (as is done in `cdrom.d`).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
```

continues

```
dtrace:::BEGIN
{
        printf("%-20s %1s %4s %6s %3s %s\n", "TIME", "D", "KB",
            "ms", "ERR", "PATH");
}

fbt::pcfs_read:entry, fbt::pcfs_write:entry, fbt::pcfs_readdir:entry
{
        self->path = args[0]->v_path;
        self->kb = args[1]->uio_resid / 1024;
        self->start = timestamp;
}

fbt::pcfs_read:return, fbt::pcfs_write:return, fbt::pcfs_readdir:return
/self->start/
{
        this->iotime = (timestamp - self->start) / 1000000;
        this->dir = probefunc == "pcfs_read" ? "R" : "W";
        printf("%-20Y %1s %4d %6d %3d %s\n", walltimestamp,
            this->dir, self->kb, this->iotime, arg1,
            self->path != NULL ? stringof(self->path) : "<null>");
        self->start = 0; self->path = 0; self->kb = 0;
}
```

Script pcfsrw.d

This script prints basic information. To retrieve `pcfs`-specific information such as the FAT type, the `struct pcfs` can be retrieved from the vnode in the same way as at the start of the `pcfs_read()` function (see the source, including `VFSTOPCFS`). We've resisted including an example of this, since `struct pcfs` has changed between Solaris versions, and it would make this script much more fragile; add the appropriate code for your Solaris version.

HSFS Scripts

HSFS is the High Sierra File System (ISO 9660) driver on Oracle Solaris, used by CD-ROMs. In cases of unusual performance or errors such as failing to mount, DTrace can be used to examine the internal operation of the device driver using the fbt provider. On recent versions of Oracle Solaris, the kernel engineers have also placed sdt provider probes in hsfs for convenience:

```
solaris# dtrace -ln 'sdt:hsfs::'
   ID   PROVIDER          MODULE                          FUNCTION NAME
83019        sdt            hsfs                 hsched_enqueue_io hsfs_io_enqueued
83020        sdt            hsfs            hsched_invoke_strategy hsfs_coalesced_io_
done
83021        sdt            hsfs            hsched_invoke_strategy hsfs_coalesced_io_
start
83022        sdt            hsfs            hsched_invoke_strategy hsfs_io_dequeued
83023        sdt            hsfs            hsched_invoke_strategy hsfs_deadline_expiry
83024        sdt            hsfs                      hsfs_getpage hsfs_compute_ra
83025        sdt            hsfs                     hsfs_getapage hsfs_io_done
83026        sdt            hsfs                     hsfs_getapage hsfs_io_wait
```

```
83027        sdt          hsfs                  hsfs_getpage_ra hsfs_readahead
83028        sdt          hsfs                     hsfs_ra_task hsfs_io_done_ra
83029        sdt          hsfs                     hsfs_ra_task hsfs_io_wait_ra
83030        sdt          hsfs                       hs_mountfs rootvp-failed
83031        sdt          hsfs                       hs_mountfs mount-done
[...]
```

The `*_ra` probes shown previously refer to read-ahead, a feature of the hsfs driver to request data ahead of time to prewarm the cache and improve performance (similar to UFS read-ahead).

Since there is currently no HSFS provider, the options are to use the fbt provider to examine driver internals; use the sdt provider (if present), because it has probe locations that were deliberately chosen for tracing with DTrace; or use the stable io provider by filtering on the CD-ROM device. For robust scripts, the best option is the io provider; the others instrument a particular operating system and version and may require modifications to match the software version you are using.

cdrom.d

The `cdrom.d` script traces the `hs_mountfs()` call via the fbt provider, showing hsfs mounts along with the mount path, error status, and mount time.

Script

The `-Z` option is used on line 1 because the hsfs driver may not yet be loaded, and the functions to probe may not yet be in memory. Once a CD-ROM is inserted, the hsfs driver is automounted.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           trace("Tracing hsfs (cdrom) mountfs...\n");
9   }
10
11  fbt::hs_mountfs:entry
12  {
13          printf("%Y:  Mounting %s... ", walltimestamp, stringof(arg2));
14          self->start = timestamp;
15  }
16
17  fbt::hs_mountfs:return
18  /self->start/
19  {
20          this->time = (timestamp - self->start) / 1000000;
21          printf("result: %d%s, time: %d ms\n", arg1,
22              arg1 ? "" : " (SUCCESS)", this->time);
```

continues

```
23          self->start = 0;
24  }
```

Script cdrom.d

Example

Here's a CD-ROM with the label "Photos001" inserted:

```
solaris# cdrom.d
Tracing hsfs (cdrom) mountfs...
2010 Jun 20 23:40:59:  Mounting /media/Photos001... result: 0 (SUCCESS), time: 157 ms
```

Several seconds passed between CD-ROM insertion and the mount initiating, as shown by `cdrom.d`. This time can be understood with more DTrace.

For example, the operation of volume management and hardware daemons can be traced (`vold(1M)`, `rmvolmgr(1M)`, `hald(1M)`, ...). Try starting this investigation with process execution:

```
solaris# dtrace -qn 'proc:::exec-success { printf("%Y %s\n", walltimestamp,
curpsinfo->pr_psargs); }'
2010 Jun 21 23:51:48 /usr/lib/hal/hald-probe-storage --only-check-for-media
2010 Jun 21 23:51:48 /usr/lib/hal/hald-probe-volume
2010 Jun 21 23:51:50 /usr/lib/hal/hal-storage-mount
2010 Jun 21 23:51:50 /sbin/mount -F hsfs -o nosuid,ro /dev/dsk/c0t0d0s2 /media/Photos0
01
2010 Jun 21 23:51:50 mount -o nosuid,ro /dev/dsk/c0t0d0s2 /media/Photos001
^C
```

The same CD-ROM was reinserted, and the HAL processes that executed to mount the CD-ROM can now be seen. DTrace can be used to further examine whether these events were triggered by a hardware interrupt (media insertion) or by polling.

UDFS Scripts

UDFS is the Universal Disk Format file system driver on Oracle Solaris, used by DVDs. This driver can be examined using DTrace in a similar way to HSFS.

dvd.d

Since the source code functions between hsfs and udfs are similar, only three lines need to be changed to `cdrom.d` for it to trace DVDs instead:

```
        trace("Tracing udfs (dvd) mountfs...\n");
fbt::ud_mountfs:entry
fbt::ud_mountfs:return
```

The output printed for mounts is the same as `cdrom.d`.

NFS Client Scripts

Chapter 7, Network Protocols, covers tracing from the NFS server. The NFS client can also be traced, which we will cover here in this chapter because the NFS mount from a client perspective behaves like any other file system. Because of this, physical (network device) I/O to serve that file system can be traced by the io provider (currently Oracle Solaris only), just like tracing physical (storage device) I/O for a local file system.

Physical I/O is not the only I/O we can use to analyze NFS client performance. Logical I/O to the NFS client driver is also interesting and may be served without performing network I/O to the NFS server—for example, when returning data from a local NFS client cache.

For kernel-based NFS drivers, all internals can be examined using the fbt provider. fbt instruments a particular operating system and version, so these scripts may therefore require modifications to match the software version you are using.

nfswizard.d

This script from the DTraceToolkit demonstrates using the io provider on Oracle Solaris to trace and summarize NFS client I/O. It traces back-end I/O only: those that trigger NFS network I/O. More I/O may be performed to the NFS share from the client, which is returned from the client cache only.

Script

This is a neat example of how you can produce a sophisticated report from basic D syntax:

```
#!/usr/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
        scriptstart = walltimestamp;
        timestart = timestamp;
}
```

continues

```

io:nfs::start
{
        /* tally file sizes */
        @file[args[2]->fi_pathname] = sum(args[0]->b_bcount);

        /* time response */
        start[args[0]->b_addr] = timestamp;

        /* overall stats */
        @rbytes = sum(args[0]->b_flags & B_READ ? args[0]->b_bcount : 0);
        @wbytes = sum(args[0]->b_flags & B_READ ? 0 : args[0]->b_bcount);
        @events = count();
}

io:nfs::done
/start[args[0]->b_addr]/
{
        /* calculate and save response time stats */
        this->elapsed = timestamp - start[args[0]->b_addr];
        @maxtime = max(this->elapsed);
        @avgtime = avg(this->elapsed);
        @qnztime = quantize(this->elapsed / 1000);
}

dtrace:::END
{
        /* print header */
        printf("NFS Client Wizard. %Y -> %Y\n\n", scriptstart, walltimestamp);

        /* print read/write stats */
        printa("Read:  %@d bytes ", @rbytes);
        normalize(@rbytes, 1000000);
        printa("(%@d Mb)\n", @rbytes);
        printa("Write: %@d bytes ", @wbytes);
        normalize(@wbytes, 1000000);
        printa("(%@d Mb)\n\n", @wbytes);

        /* print throughput stats */
        denormalize(@rbytes);
        normalize(@rbytes, (timestamp - timestart) / 1000000);
        printa("Read:  %@d Kb/sec\n", @rbytes);
        denormalize(@wbytes);
        normalize(@wbytes, (timestamp - timestart) / 1000000);
        printa("Write: %@d Kb/sec\n\n", @wbytes);

        /* print time stats */
        printa("NFS I/O events:    %@d\n", @events);
        normalize(@avgtime, 1000000);
        printa("Avg response time: %@d ms\n", @avgtime);
        normalize(@maxtime, 1000000);
        printa("Max response time: %@d ms\n\n", @maxtime);
        printa("Response times (us):%@d\n", @qnztime);

        /* print file stats */
        printf("Top 25 files accessed (bytes):\n");
        printf("   %-64s %s\n", "PATHNAME", "BYTES");
        trunc(@file, 25);
        printa("   %-64s %@d\n", @file);
}
```

Script nfswizard.d

The io provider is used to trace client NFS I/O only, by including `nfs` in the probe module field. This is technically an unstable field of the probe name, although it's also unlikely to be renamed any time soon. An alternate approach would be to trace all io probes and use a predicate to match when `args[1]->dev_name` was equal to `nfs`. See the io provider description in Chapter 4 for more discussion about matching this field for io probes.

Example

Here `nfswizard.d` was run for a few seconds while a `tar(1)` command archived files from an NFSv4 share:

```
client# nfswizard.d
Tracing... Hit Ctrl-C to end.
^C
NFS Client Wizard. 2010 Jun 22 05:32:23 -> 2010 Jun 22 05:32:26

Read:  56991744 bytes (56 Mb)
Write: 0 bytes (0 Mb)

Read:  18630 Kb/sec
Write: 0 Kb/sec

NFS I/O events:    1747
Avg response time: 2 ms
Max response time: 59 ms

Response times (us):
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |@@@@@                                    221
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         1405
            2048 |@                                        37
            4096 |                                         21
            8192 |@                                        31
           16384 |                                         19
           32768 |                                         12
           65536 |                                         0

Top 25 files accessed (bytes):
   PATHNAME                                                     BYTES
   /net/mars/export/home/brendan/Downloads/ping.tar             40960
   /net/mars/export/home/brendan/Downloads/pkg_get.pkg          69632
   /net/mars/export/home/brendan/Downloads/procps-3.2.8.tar.gz  286720
   /net/mars/export/home/brendan/Downloads/psh-i386-40          2260992
   /net/mars/export/home/brendan/Downloads/proftpd-1.3.2c.tar.gz 3174400
   /net/mars/export/home/brendan/Downloads/perlsrc-5.8.8stable.tar 51159040
```

The output includes a distribution plot of response times, which includes network latency and NFS server latency—which may return from cache (fast) or disk (slow), depending on the I/O.

nfs3sizes.d

This script shows both logical (local) and physical (network) reads by an Oracle Solaris NFSv3 client, showing requested read size distributions and total bytes. It can be used as a starting point to investigate.

> **Client caching**: The nfs client driver performs caching (unless it is directed not to, such as with the `forcedirectio` mount option), meaning that many of the logical reads may return from the client's DRAM without performing a (slower) NFS read to the server.
>
> **Read size**: The nfs client driver read size may differ from the application read size on NFS files (this can be tuned to a degree using the `rsize` mount option).

Script

The `nfs3_read()` function is the VFS interface into the NFSv3 client driver, which is traced to show requested NFS reads. The `nfs3_getpage()` and `nfs3_directio_read()` functions perform NFSv3 network I/O.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing NFSv3 client file reads... Hit Ctrl-C to end.\n");
}

fbt::nfs3_read:entry
{
        @q["NFS read size (bytes)"] = quantize(args[1]->uio_resid);
        @s["NFS read (bytes)"] = sum(args[1]->uio_resid);
}

fbt::nfs3_directio_read:entry
{
        @q["NFS network read size (bytes)"] = quantize(args[1]->uio_resid);
        @s["NFS network read (bytes)"] = sum(args[1]->uio_resid);
}

fbt::nfs3_getpage:entry
{
        @q["NFS network read size (bytes)"] = quantize(arg2);
        @s["NFS network read (bytes)"] = sum(arg2);
}
```

Script nfs3sizes.d

This script traces the size of read requests. The size of the returned data may be smaller than was requested, or zero if the read failed; the script could be enhanced to trace the returned data size instead if desired.

Example

An application performed random 1KB reads on a file shared over NFSv3:

```
client# nfssizes.d
Tracing NFSv3 client file reads... Hit Ctrl-C to end.
^C

  NFS network read size (bytes)
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2564
            8192 |                                         2
           16384 |                                         0

  NFS read size (bytes)
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 147083
            2048 |                                         0

  NFS network read (bytes)                                    10518528
  NFS read (bytes)                                           150613423
```

In this example, there were many more logical NFS reads (147,084) than physical network reads (2,566) to the NFS server, suggesting that the NFS client cache is serving most of these logical reads (high client cache hit rate). The difference between logical and physical read size distribution can also be compared, which shows that the nfs client driver is requesting 4+KB reads to satisfy 1+KB requests. Both of these behaviors can be investigated further by DTracing more internals from the nfs client driver.

nfs3fileread.d

This script shows both logical and physical (network) reads by an Oracle Solaris NFSv3 client, showing the requested and network read bytes by filename. This is a variation of the `nfs3sizes.d` script explained previously.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing NFSv3 client file reads... Hit Ctrl-C to end.\n");
}

```

continues

```
fbt::nfs3_read:entry
{
        this->path = args[0]->v_path;
        this->bytes = args[1]->uio_resid;
        @r[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::nfs3_directio_read:entry
{
        this->path = args[0]->v_path;
        this->bytes = args[1]->uio_resid;
        @n[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

fbt::nfs3_getpage:entry
{
        this->path = args[0]->v_path;
        this->bytes = arg2;
        @n[this->path ? stringof(this->path) : "<null>"] = sum(this->bytes);
}

dtrace:::END
{
        printf(" %-56s %10s %10s\n", "FILE", "READ(B)", "NET(B)");
        printa(" %-56s %@10d %@10d\n", @r, @n);
}
```

Script nfs3fileread.d

Example

All of the files read were 10MB in size and were read sequentially.

```
client# nfs3fileread.d
Tracing NFSv3 client file reads... Hit Ctrl-C to end.
^C
 FILE                                                        READ(B)     NET(B)
 /saury-data-0/10m_d                                         4182016    1265216
 /saury-data-0/10m_a                                        10493952          0
 /saury-data-0/10m_c                                        10493952   10485760
 /saury-data-0/10m_b                                        43753984   10485760
```

The difference between the `READ` (requested read bytes) and `NET` (network read bytes) columns are because of the following.

- `10m_d`: About 4MB was read from this file, which was partially cached.
- `10m_a`: This file was entirely cached in the client's DRAM and was read through once.
- `10m_c`: This file was entirely uncached and was read through once from the NFS server.
- `10m_b`: This file was entirely uncached and was read through multiple times—the first reading it from the NFS server.

TMPFS Scripts

tmpfs is a file system type for temporary files that attempts to reside in memory for fast access. It's used by Oracle Solaris for `/tmp` and other directories. The performance of `/tmp` can become a factor when tmpfs contains more data than can fit in memory, and it begins paging to the swap devices.

tmpfs activity can be traced at other levels such as the syscall interface and VFS. The scripts in this section demonstrate examining activity from the kernel tmpfs driver, using the fbt provider. fbt instruments a particular operating system and version, so these scripts may therefore require modifications to match the software version you are using. You shouldn't have too much difficulty rewriting them to trace at syscall or VFS instead if desired and to match only activity to `/tmp` or tmpfs.

tmpusers.d

This script shows who is using tmpfs on Oracle Solaris by tracing the user, process, and filename for tmpfs open calls.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      printf("%6s %6s %-16s %s\n", "UID", "PID", "PROCESS", "FILE");
}

fbt::tmp_open:entry
{
      printf("%6d %6d %-16s %s\n", uid, pid, execname,
          stringof((*args[0])->v_path));
}
```

Script tmpusers.d

Example

Here's an example:

```
solaris# tmpusers.d
   UID    PID PROCESS          FILE
     0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
     0     47 svc.configd      /etc/svc/volatile
     0     47 svc.configd      /etc/svc/volatile/sqlite_UokyAO1gmAy2L8H
     0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
```

continues

```
      0     47 svc.configd      /etc/svc/volatile
      0     47 svc.configd      /etc/svc/volatile/sqlite_Ws9dGwSvZRtutXk
      0     47 svc.configd      /etc/svc/volatile/svc_nonpersist.db-journal
      0     47 svc.configd      /etc/svc/volatile/sqlite_zGn0Ab6VUI6IFpr
[...]
      0   1367 sshd             /etc/svc/volatile/etc/ssh/sshd_config
      0   1368 sshd             /var/run/sshd.pid
```

tmpgetpage.d

This script shows which processes are actively reading from tmpfs files by tracing the tmpfs getpage routine, which is the interface to read pages of data. The time spent in `getpage` is shown as a distribution plot.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      trace("Tracing tmpfs disk read time (us):\n");
}

fbt::tmp_getpage:entry
{
      self->vp = args[0];
      self->start = timestamp;
}

fbt::tmp_getpage:return
/self->start/
{
      @[execname, stringof(self->vp->v_path)] =
          quantize((timestamp - self->start) / 1000);
      self->vp = 0;
      self->start = 0;
}
```

Script tmpgetpage.d

Example

Here the `cksum(1)` command was reading a file that was partially in memory. The time for `getpage` shows two features: fast I/O between 0 us and 4 us and slower I/O mostly between 128 us and 1024 us. These are likely to correspond to reads from DRAM or from disk (swap device). If desired, the script could be enhanced to trace disk I/O calls so that a separate distribution plot could be printed for DRAM reads and disk reads.

```
solaris# tmpgetpage.d
Tracing tmpfs disk read time (us):
^C

  cksum                                         /tmp/big0

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@                     9876
               2 |@@@@@@@@@@                               5114
               4 |                                         29
               8 |                                         48
              16 |@                                        354
              32 |                                         120
              64 |                                         19
             128 |@                                        317
             256 |@@@@@@@                                  3223
             512 |@                                        444
            1024 |                                         71
            2048 |                                         31
            4096 |                                         37
            8192 |                                         33
           16384 |                                         23
           32768 |                                         4
           65536 |                                         2
          131072 |                                         0
```

Case Study

Here we present the application of the DTrace commands, scripts, and methods discussed in this chapter.

ZFS 8KB Mirror Reads

This case study looks at a ZFS workload doing 8KB reads from a mirrored zpool.

System:
- **7410**: 4 AMD Barcelona CPUs, 128GB DRAM, one 10Gb Ethernet port
- **1 JBOD**: 22 1TB disks, 2 Logzillas, mirroring
- **ZFS**: 10 shares, 8KB record size

Workload:
- NFSv3 streaming reads, 1MB I/O size
- 100 threads total, across 10 clients (10 threads per client)
- 200+GB working set, mostly uncached

Clients:
- 10 blades

Total throughput for this workload is 338MB/sec. The 10Gb Ethernet port has a theoretical maximum throughput of 1.16GB/sec, so what is holding us back? Disk I/O latency? CPU?

Basic Observability

Operating system tools commonly used to check system performance include `vmstat(1M)`, `mpstat(1M)`, and `iostat(1M)`. Running these

```
# vmstat 5
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s6 s7 s1 s1   in   sy   cs us sy id
 0 0 0 129657948 126091808 13 13 0 0 0 2  4  4 19  3 3088 2223  990  0  1 99
 8 0 0 7527032 3974064 0 42  0  0  0  0  0  2  1  0 303 570205 2763 100141 0 62 37
 7 0 0 7527380 3974576 0  7  0  0  0  0  0  0  0  0 309 561541 2613 99200 0 62 38
 6 0 0 7526472 3973564 0  4  0  0  0  0  0  0  0  0 321 565225 2599 101515 0 62 37
 7 0 0 7522756 3970040 11 85 0  0  0  0  0  7  7  0 324 573568 2656 99129 0 63 37
[...]
```

`vmstat(1M)` shows high sys time (62 percent).

```
# mpstat 5
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
[...summary since boot truncated...]
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 21242 34223  205 5482    2 1669 7249    0    28    0  58   0  42
  1    0   0 27446 30002  113 4574    2 1374 7029    0  1133    1  53   0  46
  2    0   0 198422 31967 2951 20938    3  213 2655    0    27    0  97   0   3
  4    0   0 16970 39203 3082 3866    9  829 6695    0    55    0  59   0  40
  5    4   0 24698 33998   10 5492    3 1066 7492    0    43    0  57   0  43
  6    0   0 26184 41790   11 7412    1 1568 6586    0    15    0  67   0  33
  7   14   0 17345 41765    9 4797    1  943 5764    1    98    0  65   0  35
  8    5   0 17756 36776   37 6110    4 1183 7698    0    62    0  58   0  41
  9    0   0 17265 31631    9 4608    2  877 7784    0    37    0  53   0  47
 10    2   0 24961 34622    7 5553    1 1022 7057    0   109    1  57   0  42
 11    3   0 33744 40631   11 8501    4 1742 6755    0    72    1  65   0  35
 12    2   0 27320 42180  468 7710   18 1620 7222    0   381    0  65   0  35
 13    1   0 20360 63074 15853 5154   28 1095 6099    0    36    1  72   0  27
 14    1   0 13996 31832    9 4277    8  878 7586    0    36    0  52   0  48
 15    8   0 19966 36656    5 5646    7 1054 6703    0   633    2  56   0  42
[...]
```

`mpstat(1M)` shows CPU 2 is hot at 97 percent sys, and we have frequent cross calls (`xcals`), especially on CPU 2.

```
# iostat -xnz 5
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...summary since boot truncated...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.2   23.4   12.8 1392.7  0.5  0.1   20.3    2.3   6   5 c3t0d0
```

```
    0.0   22.4    0.0 1392.7  0.5  0.0   22.3    1.7   6   4 c3t1d0
  324.8    0.0 21946.8    0.0  0.0  4.7    0.0   14.4   1  79 c4t5000C5001073ECF5d0
  303.8    0.0 19980.0    0.0  0.0  4.0    0.0   13.1   1  75 c4t5000C50010741BF9d0
  309.8    0.0 22036.5    0.0  0.0  5.3    0.0   17.0   1  82 c4t5000C5001073ED34d0
  299.6    0.0 19944.1    0.0  0.0  4.4    0.0   14.7   1  76 c4t5000C5000D416FFEd0
  302.6    0.0 20229.0    0.0  0.0  4.4    0.0   14.4   1  77 c4t5000C50010741A8Ad0
  292.2    0.0 19198.3    0.0  0.0  4.0    0.0   13.8   1  74 c4t5000C5000D416E2Ed0
  305.6    0.0 21203.4    0.0  0.0  4.5    0.0   14.8   1  80 c4t5000C5001073DEB9d0
  280.8    0.0 18160.5    0.0  0.0  4.0    0.0   14.3   1  75 c4t5000C5001073E602d0
  304.2    0.0 19574.9    0.0  0.0  4.3    0.0   14.2   1  77 c4t5000C50010743CFAd0
  322.0    0.0 21906.5    0.0  0.0  5.1    0.0   15.8   1  80 c4t5000C5001073F2F8d0
  295.8    0.0 20115.4    0.0  0.0  4.6    0.0   15.7   1  77 c4t5000C5001073F440d0
  289.2    0.0 20836.0    0.0  0.0  4.6    0.0   16.0   1  75 c4t5000C5001073E2F4d0
  278.6    0.0 18159.2    0.0  0.0  3.8    0.0   13.6   1  73 c4t5000C5001073D840d0
  286.4    0.0 21366.9    0.0  0.0  5.0    0.0   17.5   1  79 c4t5000C5001073ED40d0
  307.6    0.0 19198.1    0.0  0.0  4.2    0.0   13.5   1  74 c4t5000C5000D416F21d0
  292.4    0.0 19045.3    0.0  0.0  4.2    0.0   14.2   1  76 c4t5000C5001073E593d0
  293.2    0.0 20590.0    0.0  0.0  5.2    0.0   17.7   1  81 c4t5000C50010743BD1d0
  317.2    0.0 21036.5    0.0  0.0  3.9    0.0   12.4   1  74 c4t5000C5000D416E76d0
  295.6    0.0 19540.1    0.0  0.0  4.0    0.0   13.5   1  72 c4t5000C5001073DDB4d0
  332.6    0.0 21610.2    0.0  0.0  4.2    0.0   12.5   1  75 c4t5000C500106CF55Cd0
[...]
```

iostat(1M) shows the disks are fairly busy (77 percent).

Just based on this information, there is little we can do to improve performance except upgrade to faster CPUs and faster disks. We could also check kernel tuning parameters to prevent CPU 2 from running hot, but at this point we don't even know *why* it is hot. It could be the cross cals, but we can't tell for certain that they are responsible for the high sys time. Without DTrace, we've hit a brick wall.

Enter DTrace

First we'll use DTrace to check high system time by profiling kernel stacks on-CPU and for the hot CPU 2:

```
# dtrace -n 'profile-1234 { @[stack()] = count(); } tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
 11  85688                        :tick-5sec
[...output truncated...]


              unix`0xfffffffffb84fd8a
              zfs`zio_done+0x383
              zfs`zio_execute+0x89
              genunix`taskq_thread+0x1b7
              unix`thread_start+0x8
             2870

              unix`do_splx+0x80
              unix`xc_common+0x231
              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
```

continues

```
                unix`hat_unload+0x41
                unix`segkmem_free_vn+0x6f
                unix`segkmem_free+0x27
                genunix`vmem_xfree+0x104
                genunix`vmem_free+0x29
                genunix`kmem_free+0x20b
                genunix`dblk_lastfree_oversize+0x69
                genunix`dblk_decref+0x64
                genunix`freeb+0x80
                ip`tcp_rput_data+0x25a6
                ip`squeue_enter+0x330
                ip`ip_input+0xe31
                mac`mac_rx_soft_ring_drain+0xdf
               3636

                unix`mach_cpu_idle+0x6
                unix`cpu_idle+0xaf
                unix`cpu_idle_adaptive+0x19
                unix`idle+0x114
                unix`thread_start+0x8
              30741
```

This shows that we are hottest in `do_splx()`, a function used to process cross calls (see `xc_call()` further down the stack).

Now we check the hot stacks for CPU 2, by matching it in a predicate:

```
# dtrace -n 'profile-1234 /cpu == 2/ { @[stack()] = count(); }
tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  8  85688                       :tick-5sec
[...output truncated...]

                unix`do_splx+0x80
                unix`xc_common+0x231
                unix`xc_call+0x46
                unix`hat_tlb_inval+0x283
                unix`x86pte_inval+0xaa
                unix`hat_pte_unmap+0xfd
                unix`hat_unload_callback+0x193
                unix`hat_unload+0x41
                unix`segkmem_free_vn+0x6f
                unix`segkmem_free+0x27
                genunix`vmem_xfree+0x104
                genunix`vmem_free+0x29
                genunix`kmem_free+0x20b
                genunix`dblk_lastfree_oversize+0x69
                genunix`dblk_decref+0x64
                genunix`freeb+0x80
                ip`tcp_rput_data+0x25a6
                ip`squeue_enter+0x330
                ip`ip_input+0xe31
                mac`mac_rx_soft_ring_drain+0xdf
               1370
```

This shows that CPU 2 is indeed hot in cross calls. To quantify the problem, we could postprocess this output to add up which stacks are cross calls and which aren't, to calculate the percentage of time spent in cross calls.

Sometimes frequency counting the kernel function name that is on-CPU is sufficient to identify the activity, instead of counting the entire stack:

```
# dtrace -n 'profile-1234 /cpu == 2/ { @[func(arg0)] = count(); }
tick-5sec { exit(0); }'
dtrace: description 'profile-1234 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  85688                       :tick-5sec

  mac`mac_hwring_tx                                                 1
  mac`mac_soft_ring_worker_wakeup                                   1
  mac`mac_soft_ring_intr_disable                                    1
  rootnex`rootnex_init_win                                          1
  scsi_vhci`vhci_scsi_init_pkt                                      1
[...output truncated...]
  unix`setbackdq                                                   31
  ip`ip_input                                                      33
  unix`atomic_add_64                                               33
  unix`membar_enter                                                38
  unix`page_numtopp_nolock                                         47
  unix`0xfffffffffb84fd8a                                          50
  unix`splr                                                        56
  genunix`ddi_dma_addr_bind_handle                                 56
  unix`i_ddi_vaddr_get64                                           62
  unix`ddi_get32                                                   81
  rootnex`rootnex_coredma_bindhdl                                  83
  nxge`nxge_start                                                  92
  unix`mutex_delay_default                                         93
  unix`mach_cpu_idle                                              106
  unix`hat_tlb_inval                                              126
  genunix`biodone                                                 157
  unix`mutex_enter                                                410
  unix`do_splx                                                   2597
```

This output is easier to examine and still identifies the cross call samples as the hottest CPU activity (`do_splx()` function). By postprocessing the sample counts (summing the count column using `awk(1)`), we found that CPU 2 spent 46 percent of its time in `do_splx()`, which is a significant percentage of time.

Investigating Cross Calls

CPU cross calls can be probed using DTrace directly:

```
# dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); } tick-5sec { exit(0); }'
dtrace: description 'sysinfo:::xcalls ' matched 2 probes
CPU     ID                    FUNCTION:NAME
 10  85688                       :tick-5sec
[...output truncated...]

              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
```

continues

```
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freemsg+0x84
              nxge`nxge_txdma_reclaim+0x396
              nxge`nxge_start+0x327
              nxge`nxge_tx_ring_send+0x69
              mac`mac_hwring_tx+0x20
              mac`mac_tx_send+0x262
              mac`mac_tx_soft_ring_drain+0xac
           264667

              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              genunix`kmem_free+0x20b
              genunix`dblk_lastfree_oversize+0x69
              genunix`dblk_decref+0x64
              genunix`freeb+0x80
              ip`tcp_rput_data+0x25a6
              ip`squeue_enter+0x330
              ip`ip_input+0xe31
              mac`mac_rx_soft_ring_drain+0xdf
              mac`mac_soft_ring_worker+0x111
              unix`thread_start+0x8
           579607
```

The most frequent stacks originate in either `ip` (the IP and TCP module) or `nxge` (which is the 10GbE network interface driver). Filtering on CPU 2 (`/cpu == 2/`) showed the same hottest stacks for these cross calls. Reading up the stack to understand the nature of these cross calls shows that they enter the kernel memory subsystem (*Solaris Internals* [McDougall and Mauro, 2006] is a good reference for understanding these).

Perhaps the most interesting stack line is `dblk_lastfree_oversize()`—oversize is the kernel memory allocator slab for large buffers. Although it is performing well enough, the other fixed-size slabs (8KB, 64KB, 128KB, and so on) perform better, so usage of oversize is undesirable if it can be avoided.

The cross call itself originates from a code path that is freeing memory, including functions such as `kmem_free()`. To better understand this cross call, the `kmem_free()` function is traced so that the size freed can be examined if this becomes a cross call on CPU 2:

```
# dtrace -n 'fbt::kmem_free:entry /cpu == 2/ { self->size = arg1; }
sysinfo:::xcalls /cpu == 2/ { @ = quantize(self->size); }'
dtrace: description 'fbt::kmem_free:entry ' matched 2 probes
^C


           value  ------------- Distribution ------------- count
          524288 |                                         0
         1048576 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 37391
         2097152 |                                         0
```

The output shows that the frees that become cross calls are in the 1MB to 2MB range.

This rings a bell. The clients are using a 1MB I/O size for their sequential reads, on the assumption that 1MB would be optimal. Perhaps it is these 1MB I/Os that are causing the use of the oversize kmem cache and the cross calls.

Trying the Solution

As an experiment, we changed I/O size on the clients to 128KB. Let's return to system tools for comparison:

```
# mpstat 5
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
[...summary since boot truncated...]
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0 2478  7196  205 10189    2 2998 3934    0    41    0  47   0  53
  1    0   0  139  6175  111 9367    2 2713 3714    0    84    0  44   0  56
  2    0   0 10107 11434 3610 54281   11 1476 2329    0   465    1  79   0  20
  4    7   0   36  7924 3703 6027   11 1412 5085    0   146    1  54   0  46
  5    0   0    4  5647   10 8028    3 1793 4347    0    28    0  53   0  47
  6    1   0   49  6984   12 12248    2 2863 4374    0    38    0  56   0  44
  7    0   0   11  4472   10 7687    3 1730 3730    0    33    0  49   0  51
  8    0   0   82  5686   42 9783    2 2132 5116    0   735    1  49   0  49
  9    0   0   39  4339    7 7308    1 1511 4898    0   396    1  43   0  57
 10    0   0    3  5256    4 8997    1 1831 4399    0    22    0  43   0  57
 11    0   0    5  7865   12 13900    1 3080 4365    1    43    0  55   0  45
 12    0   0   58  6990  143 12108   12 2889 5199    0   408    1  56   0  43
 13    1   0    0 35884 32388 6724   48 1536 4032    0    77    0  73   0  27
 14    1   0   14  3936    9 6473    6 1472 4822    0   102    1  42   0  58
 15    3   0    8  5025    8 8460    8 1784 4360    0   105    2  42   0  56
[...]
```

The cross calls have mostly vanished, and throughput is 503MB/sec—a 49 percent improvement!

```
# iostat -xnz 5
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...summary since boot truncated...
                    extended device statistics
```

continues

```
     r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
     0.2   45.6   12.8 3982.2  1.7  0.2   37.6    4.3  19  20 c3t0d0
     0.4   45.2   25.6 3982.2  1.3  0.1   28.7    2.9  15  13 c3t1d0
   381.8    0.0 21210.8    0.0  0.0  5.2    0.0   13.7   1  92 c4t5000C5001073ECF5d0
   377.2    0.0 21914.8    0.0  0.0  5.5    0.0   14.6   1  87 c4t5000C50010741BF9d0
   330.2    0.0 21334.7    0.0  0.0  6.4    0.0   19.3   1  89 c4t5000C5001073ED34d0
   379.8    0.0 21294.8    0.0  0.0  5.4    0.0   14.3   1  92 c4t5000C5000D416FFEd0
   345.8    0.0 21823.1    0.0  0.0  6.1    0.0   17.6   1  90 c4t5000C50010741A8Ad0
   360.6    0.0 20126.3    0.0  0.0  5.2    0.0   14.5   1  85 c4t5000C5000D416E2Ed0
   352.2    0.0 23318.3    0.0  0.0  6.9    0.0   19.7   1  93 c4t5000C5001073DEB9d0
   253.8    0.0 21745.3    0.0  0.0 10.0    0.0   39.3   0 100 c4t5000C5001073E602d0
   337.4    0.0 22797.5    0.0  0.0  7.1    0.0   20.9   1  96 c4t5000C50010743CFAd0
   346.0    0.0 22145.4    0.0  0.0  6.7    0.0   19.3   1  87 c4t5000C5001073F2F8d0
   350.0    0.0 20946.2    0.0  0.0  5.3    0.0   15.2   1  89 c4t5000C5001073F440d0
   383.6    0.0 22688.1    0.0  0.0  6.5    0.0   17.0   1  94 c4t5000C5001073E2F4d0
   333.4    0.0 24451.0    0.0  0.0  8.2    0.0   24.6   1  98 c4t5000C5001073D840d0
   337.6    0.0 21057.5    0.0  0.0  5.9    0.0   17.4   1  90 c4t5000C5001073ED40d0
   370.8    0.0 21949.1    0.0  0.0  5.3    0.0   14.2   1  88 c4t5000C5000D416F21d0
   393.2    0.0 22121.6    0.0  0.0  5.6    0.0   14.3   1  90 c4t5000C5001073E593d0
   354.4    0.0 22323.5    0.0  0.0  6.4    0.0   18.1   1  93 c4t5000C50010743BD1d0
   382.2    0.0 23451.7    0.0  0.0  5.9    0.0   15.3   1  95 c4t5000C5000D416E76d0
   357.4    0.0 22791.5    0.0  0.0  6.8    0.0   19.0   1  93 c4t5000C5001073DDB4d0
   338.8    0.0 22762.6    0.0  0.0  7.3    0.0   21.6   1  92 c4t5000C500106CF55Cd0
[...]
```

The disks are now reaching 100 percent busy and have become the new bottleneck (one disk in particular). This often happens with performance investigations: As soon as one problem has been fixed, another one becomes apparent.

Analysis Continued

From the previous `iostat(1M)` output, it can be calculated that the average I/O size is fairly large (~60KB), yet this results in low throughput per disk (20MB/sec) for disks that can pull more than 80MB/sec. This could indicate a random component to the I/O. However, with DTrace, we can measure it directly.

Running `bitesize.d` from Chapter 4 (and the DTraceToolkit) yields the following:

```
# bitesize.d
Tracing... Hit Ctrl-C to end.

     PID  CMD
    1040  /usr/lib/nfs/nfsd -s /var/ak/rm/pool-0/ak/nas/nfs4\0

           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 8296
           16384 |                                         0

       0  sched\0

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |                                         8
            1024 |                                         51
            2048 |                                         65
            4096 |                                         25
            8192 |@@@@@@@@                                 5060
```

```
          16384 |@@@@                                     2610
          32768 |@@@@                                     2881
          65536 |@@@@@@@@@@@@@                            8576
         131072 |@@@@@@@@@@@                              7389
         262144 |                                         0
```

This shows I/O from 8KB through to 128KB. 8KB I/O is expected because of the ZFS record size and when nfsd responds to a request by reading 8KB I/O. Doing this sequentially will trigger ZFS prefetch, which will read ahead in the file asynchronously to the nfsd thread (`sched`). The vdev layer can aggregate these reads up to 128KB before they are sent to disk. All of these internals can be examined using DTrace.

Running `seeksize.d` from Chapter 4 (and the DTraceToolkit) yields the following:

```
# seeksize.d
Tracing... Hit Ctrl-C to end.

     PID  CMD
    1040  /usr/lib/nfs/nfsd -s /var/ak/rm/pool-0/ak/nas/nfs4\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@                 5387
               1 |                                         1
               2 |                                         53
               4 |                                         3
               8 |                                         7
              16 |@@                                       450
              32 |@@                                       430
              64 |@                                        175
             128 |@                                        161
             256 |@                                        144
             512 |                                         97
            1024 |                                         49
            2048 |                                         10
            4096 |                                         19
            8192 |                                         34
           16384 |                                         84
           32768 |@                                        154
           65536 |@                                        307
          131072 |@@                                       528
          262144 |@@@                                      598
          524288 |@                                        266
         1048576 |                                         23
         2097152 |                                         0

       0  sched\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@                                    3160
               1 |                                         2
               2 |                                         38
               4 |                                         11
               8 |                                         3
              16 |                                         265
              32 |@                                        309
```

continues

```
              64 |@                                        442
             128 |@                                        528
             256 |@                                        767
             512 |@                                        749
            1024 |@                                        427
            2048 |                                         165
            4096 |                                         250
            8192 |@                                        406
           16384 |@                                        870
           32768 |@@@                                      1623
           65536 |@@@@@                                    2801
          131072 |@@@@@@@                                  4113
          262144 |@@@@@@@                                  4167
          524288 |@@@                                      1787
         1048576 |                                         141
         2097152 |                                         7
         4194304 |@                                        718
         8388608 |@                                        354
        16777216 |                                         0
```

This shows that the disks are often seeking to perform I/O. From this, we could look at how the files were created and what file system parameters existed to optimize placement in order to reduce seeking.

Running iopattern from Chapter 4 (and the DTraceToolkit) yields the following:

```
# iopattern
%RAN %SEQ  COUNT    MIN    MAX    AVG      KR     KW
  72   28  72996     36 131072  59152 4130835  85875
  70   30  71971    512 131072  61299 4217260  91147
  67   33  68096    512 131072  59652 3872788  94092
  63   37  72490     36 131072  60248 4173898  91155
  66   34  73607    512 131072  60835 4285085  95988
[...]
```

iopattern confirms the previous findings.

Finally, an `iolatency.d` script was written to show overall device latency as a distribution plot:

```
#!/usr/sbin/dtrace -s

io:::start
{
        start[arg0] = timestamp;
}

io:::done
/start[arg0]/
{
        @time["disk I/O latency (ns)"] = quantize(timestamp - start[arg0]);
      start[arg0] = 0;
}
```

Script iolatency.d

```
# iolatency.d -n 'tick-5sec { exit(0); }'
dtrace: script 'io-latency.d' matched 10 probes
dtrace: description 'tick-5sec ' matched 1 probe
CPU     ID                    FUNCTION:NAME
 15  85688                       :tick-5sec

  disk I/O latency (ns)
           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |                                         1
          131072 |                                         259
          262144 |@                                        457
          524288 |@@                                       1330
         1048576 |@@@@                                     2838
         2097152 |@@@@@                                    4095
         4194304 |@@@@@@@                                  5303
         8388608 |@@@@@@@@@                                7460
        16777216 |@@@@@@@                                  5538
        33554432 |@@@@                                     3480
        67108864 |@@                                       1338
       134217728 |                                         147
       268435456 |                                         3
       536870912 |                                         0
```

The latency for these disk I/Os is fairly large, often exceeding 8 ms. There are a few ways we might improve performance here.

- Tuning file system on-disk placement to promote sequential access, which should take I/O latency closer to 1 ms.
- Upgrading (or improving) caches by increasing the size of the Level 1 cache (the ARC, which is DRAM-based) or using a level-two cache (the ZFS L2ARC, which is SSD-based) to span more of the working set. The internal workings of these caches can also be examined.
- Faster disks.

Conclusion

In this case study, we've demonstrated using DTrace to solve one problem and gather data on the next. This isn't the end of the road for DTrace—we can continue to study the internals of file system on-disk placement using DTrace, as well as the workings of the level-one file system cache to hunt for suboptimalities.

Summary

In this chapter, DTrace was used to examine file system usage and internals. This was performed from different perspectives: at the system call layer, at the virtual file system (VFS) layer, and from the file system software itself. For performance investigations, at the ability to measure I/O latency from these different layers can be crucial for pinpointing the source of latency—whether that's from the file system or underlying devices. Characteristics of the file system workload were also measured, such as I/O types and filenames, to provide context for understanding what the file system is doing and why.

6

Network Lower-Level Protocols

Network I/O is processed by many different layers and protocols, including the application, the protocol libraries, the TCP/IP stack, and the network interface driver. DTrace allows you to examine the internals of each layer, tracking a packet step-by-step as it is processed from the application to the network interface. Using DTrace, you can answer questions about system network load such as the following.

What clients are requesting network I/O?

To which TCP or UDP ports?

Which processes are generating network I/O? Why?

Are packets being dropped in the TCP/IP stack? Why?

As an example, `connections` is a DTrace-based tool you can use to trace inbound network connections:

```
solaris# connections -v
TIMESTR                  UID   PID CMD           TYPE  PORT IP_SOURCE
2010 Jan  3 01:08:34       0   753 sshd           tcp    22 192.168.2.124
2010 Jan  3 01:08:41       0  1630 inetd          tcp    23 192.168.2.241
2010 Jan  3 01:08:48       0   753 sshd           tcp    22 192.168.2.241
2010 Jan  3 01:08:52       0   753 sshd           tcp    22 192.168.2.124
2010 Jan  3 01:08:59       0   753 sshd           tcp    22 192.168.2.145
2010 Jan  3 01:09:54       0  1630 inetd          tcp    79 192.168.2.145
```

Network-sniffing tools, such as `snoop` and `tcpdump`, can show what packets transmitted over the wire and can identify that TCP completed connections. However, they cannot identify which process accepted connections or provide details beyond what is in the network packet. The connections script uses DTrace to trace at the socket layer, in the context of the accepting process, to show both network and process details.

This is the first of two chapters on networking and covers the first six network stack layers: XDR, Sockets, TCP, UDP, IP, ICMP, and the physical network interface. The next chapter focuses on application-level protocols, such as HTTP and NFS (layer-seven protocols).

Capabilities

The five-layer TCP/IP model groups the Application, Presentation, and Session layers together. The seven-layer OSI model is used here (see Figure 6-1), not just because DTrace can see the Session and Presentations layers but because tracing sockets is very useful in terms of application context, as will be demonstrated in the "Scripts" section.

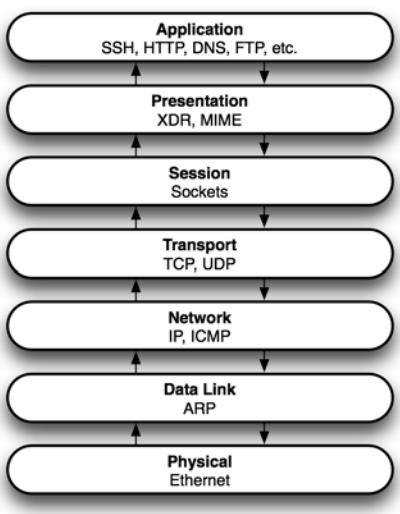


Figure 6-1 OSI model

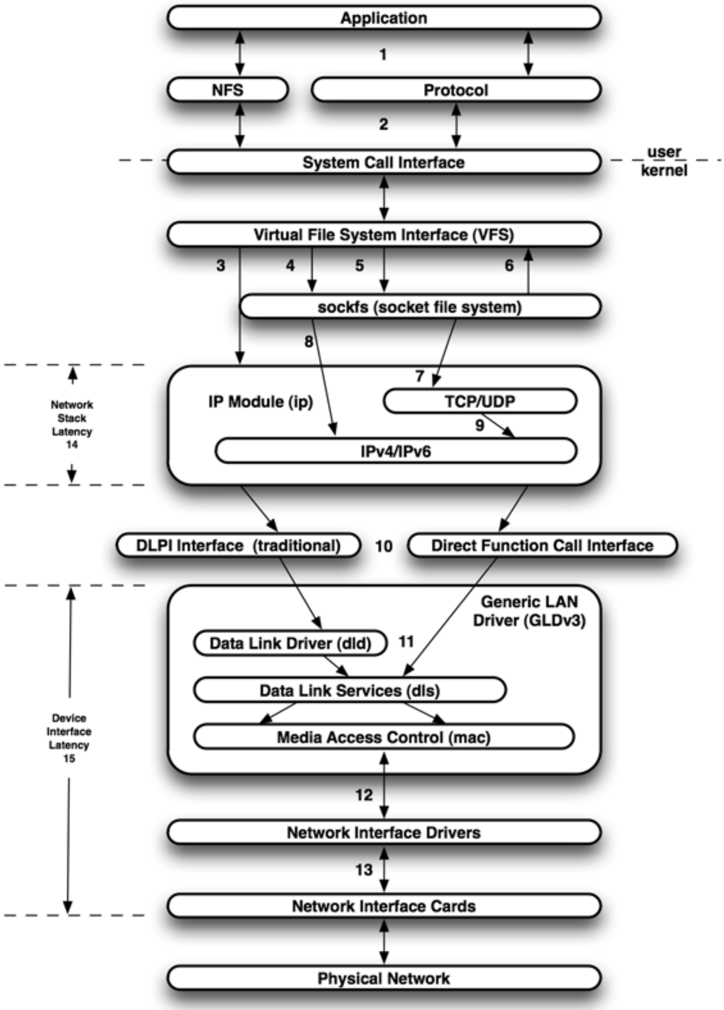


Figure 6-2 Solaris TCP/IP stack

Figure 6-2 shows the Solaris TCP/IP stack. At each of the numbered items, we can use DTrace to ask questions such as the following.

1. What protocol requests are occurring? By user stack trace? By latency?
2. What application level I/O is occurring? With protocol details?

3. What direct network I/O is occurring (for example, NFS client)?
4. What socket I/O is occurring, throughput and IOPS? By process and stack trace?
5. What socket connections are being created/accepted?
6. Is the socket layer returning errors? Why?
7. What transport I/O is occurring? TCP vs. UDP?
8. What raw network I/O is occurring? ICMP by type?
9. What IP I/O is occurring? By size? By source/destination?
10. What network interface I/O is occurring? By size?
11. What network devices are still using dld?
12. What is the frequency of driver calls?
13. Are drivers polling interfaces or waiting for interrupts?
14. What is the network stack latency?
15. What is the driver interface stack latency?

Note that application protocols, the very top of this stack (NFS, HTTP, FTP, and so on), are covered in Chapter 7, Application Protocols.

Strategy

To get started using DTrace to examine network I/O in the networking stack, follow these steps (the target of each step is in bold):

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.
2. In addition to those DTrace tools, familiarize yourself with **existing network statistic tools**. For example, `netstat -s` shows various TCP/IP statistics, `netstat -i` shows network interface statistics, and you can use `tcpdump` or `snoop` for packet details. The metrics that these print can be treated as starting points for customization with DTrace.
3. Locate or write tools to generate **known network I/O**, which could be as simple as using `ftp` to transfer a large file of a known size. Many tools exist to generate TCP and UDP I/O, including `ttcp` for simple TCP connections and `uperf` for sophisticated network I/O. It is extremely helpful to have known workloads to examine while developing DTrace scripts.
4. Customize and write your own one-liners and scripts using the **syscall provider** for socket I/O.

5. If available, try the **mib, ip, tcp, and udp providers** for writing stable one-liners and scripts.
6. To dig deeper than these providers allow, familiarize yourself with how the kernel and user-land processes call network I/O by examining **stack backtraces** (see the "fbt Provider" section). Also refer to functional diagrams of the network stack such as the OSI model shown in Figure 6-1 and the network flow diagrams. Refer to published kernel texts such as *Solaris Internals* (McDougall and Mauro, 2006) and *Mac OS X Internals* (Singh, 2006).
7. Examine kernel internals for network I/O by using the **fbt** provider, and refer to **kernel source code** (if available). Be aware that scripts using fbt may require maintenance to match updates to the kernel software.

Checklist

Consider Table 6-1 as a checklist of network I/O issue types, which can be examined using DTrace.

Table 6-1 Network I/O Checklist

Issue	Description
Volume	A server may be accepting a high volume of network I/O from unexpected clients, which may be avoidable by reconfiguring the client environment. Applications may also be performing a high volume of network I/O that could be avoided by modifying their behavior. DTrace can be used to examine network I/O by client, port, and applications to identify who is using the network, how much, and why.
Latency	There are a variety of latencies we can look at when diagnosing network I/O: **First byte latency**: The time from requesting a TCP connection to when the first byte is transferred. High latency here can be an indication of a saturated server that is taking time to create a TCP session and schedule the application. **Round-trip time**: The time for a TCP data packet to be acknowledged. High latency here can be a sign of a slow external network. **Stack latency**: The time for a packet to be processed by each layer of the TCP/IP stack. DTrace can be used to measure all of these latencies, which is essential information for understanding the performance of any application that does network I/O.

continues

Table 6-1 Network I/O Checklist (*Continued*)

Issue	Description
Queueing	I/O latency can also be caused by network I/O queuing in the network stack. Queue length and wait time such as on the TCP transmission queue and device driver queues can be examined using DTrace.
Errors	Each layer of the network stack can generate an error or misbehave in a variety of ways, such as checksum errors, packet drops, routing errors, and out-of-order packets. Some errors are not reported by user-land tools yet are known in the kernel. DTrace can be used to monitor all errors and identify whether errors are affecting applications and connections.
Configuration	There are typically many tunables that can be set to tune network performance. Are they working, and can they be tuned further? For example, DTrace can be used to check whether jumbo frames are being used and to easily identify the clients who are not using them. Other features and tunables, if available, could also be studied with DTrace: TCP window and buffer sizes, TCP Large Send Offload, TCP fusion, and so on.

Providers

Table 6-2 shows providers of interest when tracing network stack I/O.

Table 6-2 Providers for Network I/O

Provider	Description
mib	A stable provider that allows tracing when statistics for the SNMP Message Information Bases (MIBs) are incremented. Although the arguments are limited, this provider is useful for locating logical events in the code path.
ip	IPv4/IPv6 provider. Trace IP packet send and receive, with IP header details and payload length. This provider makes it easy to see which clients are connecting and what their network throughputs are. The IP provider is currently available only on recent builds of OpenSolaris.
tcp	TCP provider. Trace TCP send, receive, and connection events, with TCP header details. The TCP provider is currently only available on recent builds of OpenSolaris.
udp	UDP provider. Trace UDP send and receive with UDP header details. The UDP provider is currently available only on recent builds of OpenSolaris.
syscall	Trace entry and return of operating system calls, arguments, and return values. Much of network I/O begins as application syscalls, making this a useful provider to consider, especially as the user stack trace must be examined in application context.

Table 6-2 Providers for Network I/O (*Continued*)

Provider	Description
sched	Trace kernel scheduling events, including when threads leave CPU and return. This can be useful for following network I/O, since application threads will leave CPU when waiting for network I/O to complete and then return.
fbt	The fbt provider allows the internals of the network stack and network device drivers to be examined. This has an unstable interface and will change between releases of the operating system and drivers, meaning that scripts based on fbt may need to be significantly rewritten for each such update. The upside is that everything can be traced from fbt.
gld	Stable network providers currently in development (see the "Network Providers" section).

Check your operating system version to see which of these providers are available. At the very least, `syscall`, `fbt`, and `mib` should be available, which provide an excellent level of coverage for examining the network stack.

mib Provider

The `mib` provider traces updates to the SNMP MIB counters from the hundreds of update points in the kernel. For example, Solaris 10 5/08 probes 573 kernel locations to provide a stable interface of 212 probe names—the exported MIB counters. In the following output, the top three probes provide `mib:::ipv6IfIcmpInBadRedirects`:

```
solaris# dtrace -ln mib:::
   ID   PROVIDER  MODULE                    FUNCTION NAME
25050       mib      ip         icmp_redirect_v6 ipv6IfIcmpInBadRedirects
25051       mib      ip          icmp_inbound_v6 ipv6IfIcmpInBadRedirects
25052       mib      ip  ip_mib2_add_icmp6_stats ipv6IfIcmpInBadRedirects
25053       mib      ip        ndp_input_solicit ipv6IfIcmpInBadNeighborSolicitations
25054       mib      ip  ip_mib2_add_icmp6_stats ipv6IfIcmpInBadNeighborSolicitations
25055       mib      ip        ndp_input_advert ipv6IfIcmpInBadNeighborAdvertisements
25056       mib      ip  ip_mib2_add_icmp6_stats ipv6IfIcmpInBadNeighborAdvertisements
25057       mi       ip       ip_fanout_proto_v6 ipv6IfIcmpInOverflows
[...]
```

The probe names (`NAME`) constitute a stable interface based on SNMP MIB counters; the module and function names show the kernel implementation, which is subject to change. Since the `mib` provider accesses both in the same probes, it can be used to bridge kernel implementation with stable SNMP MIB probes, for example:

```
solaris# dtrace -n 'mib:::tcp*
{ @[strjoin(probefunc, strjoin("() -> ", probename))] = count();}'
dtrace: description 'mib:::tcp* ' matched 94 probes
^C

  tcp_connect_ipv4() -> tcpActiveOpens                                 1
  tcp_xmit_mp() -> tcpOutControl                                       1
  tcp_rput_data() -> tcpOutAck                                        12
  tcp_ack_timer() -> tcpOutAck                                        19
  tcp_ack_timer() -> tcpOutAckDelayed                                 19
  tcp_output() -> tcpOutDataBytes                                    124
  tcp_output() -> tcpOutDataSegs                                     124
  tcp_rput_data() -> tcpInAckBytes                                   124
  tcp_rput_data() -> tcpInAckSegs                                    124
  tcp_set_rto() -> tcpRttUpdate                                      124
  tcp_rput_data() -> tcpInDataInorderBytes                           146
  tcp_rput_data() -> tcpInDataInorderSegs                            146
```

For anyone considering DTracing the kernel network stack via the `fbt` provider, the `mib` provider may be used first in this way to locate functions of interest. By tracing the kernel stack (using `stack()`) instead of `probefunc`, the entire calling path can be examined.

For historical data (before DTrace was tracing), the `netstat -s` command on Solaris prints out the current value of the maintained MIB counters. These use names that closely match the probe names for the DTrace mib provider. Here's an example:

```
solaris# netstat -s
[...]
TCP     tcpRtoAlgorithm     =     4     tcpRtoMin           =   400
        tcpRtoMax           = 60000     tcpMaxConn          =    -1
        tcpActiveOpens      =3754034    tcpPassiveOpens     =145293
        tcpAttemptFails     = 16723     tcpEstabResets      = 30598
        tcpCurrEstab        =    38     tcpOutSegs          =178686476
        tcpOutDataSegs      =127818052  tcpOutDataBytes     =2924374551
        tcpRetransSegs      =686172     tcpRetransBytes     =667592163
[...]
```

The output of `netstat -s` showed a high rate of tcpRetransBytes. To understand how this occurs in the kernel TCP/IP stack, that MIB counter can be probed and the stack collected:

```
solaris# dtrace -n 'mib:::tcpRetransBytes { @[stack()] = count(); }'
dtrace: description 'mib:::tcpRetransBytes ' matched 4 probes
^C


              ip`tcp_timer_handler+0x28
              ip`squeue_drain+0xf0
              ip`squeue_worker+0xeb
              unix`thread_start+0x8
                4
```

Table 6-3 Example mib Probes

Probe	Protocol	Description
`tcpActiveOpens`	TCP	Outbound connection: fires whenever a TCP connection directly transitions from the `CLOSED` to `SYN_SENT` state.
`tcpPassiveOpens`	TCP	Inbound connection: fires whenever TCP connections directly transition from the `LISTEN` to `SYN_RCVD` state.
`tcpOutRsts`	TCP	Fires whenever a segment is sent with the RST flag set, such as for connection refused.
`tcpOutDataBytes`	TCP	Fires whenever data is sent. The number of bytes sent is in `args[0]`.
`tcpOutDataSegs`	TCP	Fires whenever a segment is sent.
`tcpInDataInorderBytes`	TCP	Fires whenever data is received such that *all* data prior to the new data's sequence number has previously been received. The number of bytes received in order is passed in `args[0]`.
`tcpInDataInorderSegs`	TCP	Fires whenever a segment is received such that *all* data prior to the new segment's sequence number has previously been received.
`udpHCOutDatagrams`	UDP	Fires whenever a UDP datagram is sent.
`udpHCInDatagrams`	UDP	Fires whenever a UDP datagram is received.
`ipIfStatsHCOutOctets`	IP	The total number of octets (bytes) sent on the interface, including framing characters.
`ipIfStatsHCInOctets`	IP	The total number of octets (bytes) received on the interface, including framing characters.

DTrace reported that this probe description matched four probes, meaning there are four locations in the kernel that increment this counter. The location that was being called has been identified in the stack trace. Those functions can now be examined in the source code and traced using the fbt provider to get a grip on exactly why this counter was incremented.

The HC in the probe names stands for High Capacity, which typically means these are 64-bit counters.

See the "`mib` Provider" chapter of the DTrace Guide for the full `mib` provider documentation and probe definitions.[1] Since these probe names are from MIBs, there are many other documentation sources for the counters, including request for comments (RFCs) that define the counters; mib definition files, such as those shipped

1. *http://wikis.sun.com/display/DTrace/mib+Provider*

in Solaris under `/etc/sma/snmp/mibs`; plus the Solaris `mib` header file `/usr/include/inet/mib2.h`.

ip Provider

The `ip` provider traces the IPv4 and IPv6 protocols. Probes and arguments for the `ip` provider are listed in Tables 6-4 and 6-5 and are also shown in the `ip` provider section of the DTrace Guide.[2]

These probes trace packets on physical interfaces as well as packets on loopback interfaces that are processed by `ip`. These can be differentiated using the `args[3]->if_local` argument in a predicate when an `ip` provider probe fires (see Table 6-5). An IP packet must have a full IP header to be visible to these probes.

> **Note**
>
> Loopback TCP packets on Solaris may be processed by *tcp fusion*, a performance feature that bypasses the ip layer. These are packets over a fused connection, which will not be visible using the `ip:::send` and `ip:::receive` probes (but they can be seen using the `tcp:::send` and `tcp:::receive` probes). When TCP fusion is enabled (which it is by default), loopback connections become fused after the TCP handshake, and then all data packets take a shorter code path that bypasses the ip layer.

Table 6-5 shows the arguments to the ip probes. These argument types are designed to be reused where possible for other network provider probes, as discussed in the "Network Providers" section.

Table 6-4 ip Provider Probes

Probe	Description
`send`	Fires whenever the kernel network stack sends an `ip` packet
`receive`	Fires whenever the kernel network stack receives an `ip` packet

Table 6-5 ip Probe Arguments

Probe	args[0]	args[1]	args[2]	args[3]	args[4]	args[5]
`send`	`pktinfo_t *`	`csinfo_t *`	`ipinfo_t *`	`ifinfo_t *`	`ipv4info_t *`	`ipv6info_t *`
`receive`	`pktinfo_t *`	`csinfo_t *`	`ipinfo_t *`	`ifinfo_t *`	`ipv4info_t *`	`ipv6info_t *`

2. *http://wikis.sun.com/display/DTrace/ip+Provider*

pktinfo_t

The pktinfo_t structure is where packet ID info can be made available for more detailed analysis. However, it is not currently implemented. The pkt_addr member is currently always NULL.

```
typedef struct pktinfo {
        uintptr_t pkt_addr;             /* currently always NULL */
} pktinfo_t;
```

Should packet IDs become available, measuring network stack layer-to-layer latency will become relatively easy, using the packet ID as a key to an associative array storing the previous layer time stamp.

csinfo_t

The csinfo_t structure is used for systemwide connection state information. It contains a unique identifier, cs_cid, which can be used as a key for an associative array, to cache data by connection, which can then be retrieved from other events. It also has cs_pid, for the process ID that created the connection.

```
typedef struct csinfo {
        uintptr_t cs_addr;
        uint64_t cs_cid;
        pid_t cs_pid;
        zoneid_t cs_zoneid;
} csinfo_t;
```

Note that the original integration (and documentation) of the ip provider had csinfo_t as a placeholder for future additions, with cs_addr as the only member (raw pointer to conn_t). At the time of writing, the additional members shown previously now exist but are populated only for the tcp and udp providers. Additional work is required for these to work for the ip provider as well.

ipinfo_t

The ipinfo_t structure contains common IP information for both IPv4 and IPv6.

```
typedef struct ipinfo {
        uint8_t ip_ver;                 /* IP version (4, 6) */
        uint16_t ip_plength;            /* payload length */
        string ip_saddr;                /* source address */
        string ip_daddr;                /* destination address */
} ipinfo_t;
```

ifinfo_t

The `ifinfo_t` structure contains network interface information.

```
typedef struct ifinfo {
        string if_name;                 /* interface name */
        int8_t if_local;                /* is delivered locally */
        netstackid_t if_ipstack;        /* ipstack ID */
        uintptr_t if_addr;              /* pointer to raw ill_t */
} ifinfo_t;
```

The `if_local` member is 1 for a local interface (loopback), 0 for not a local interface, and 1 if unknown.

ipv4info_t

The `ipv4info_t` structure is a DTrace-translated version of the IPv4 header.

```
typedef struct ipv4info {
        uint8_t ipv4_ver;                /* IP version (4) */
        uint8_t ipv4_ihl;                /* header length, bytes */
        uint8_t ipv4_tos;                /* type of service field */
        uint16_t ipv4_length;            /* length (header + payload) */
        uint16_t ipv4_ident;             /* identification */
        uint8_t ipv4_flags;              /* IP flags */
        uint16_t ipv4_offset;            /* fragment offset */
        uint8_t ipv4_ttl;                /* time to live */
        uint8_t ipv4_protocol;           /* next level protocol */
        string ipv4_protostr;            /* next level protocol, as a string */
        uint16_t ipv4_checksum;          /* header checksum */
        ipaddr_t ipv4_src;               /* source address */
        ipaddr_t ipv4_dst;               /* destination address */
        string ipv4_saddr;               /* source address, string */
        string ipv4_daddr;               /* destination address, string */
        ipha_t *ipv4_hdr;                /* pointer to raw header */
} ipv4info_t;
```

ipv6info_t

The `ipv6info_t` structure is a DTrace-translated version of the IPv6 header.

```
typedef struct ipv6info {
        uint8_t ipv6_ver;                /* IP version (6) */
        uint8_t ipv6_tclass;             /* traffic class */
        uint32_t ipv6_flow;              /* flow label */
        uint16_t ipv6_plen;              /* payload length */
        uint8_t ipv6_nexthdr;            /* next header protocol */
        string ipv6_nextstr;             /* next header protocol, as a string*/
        uint8_t ipv6_hlim;               /* hop limit */
        in6_addr_t *ipv6_src;            /* source address */
        in6_addr_t *ipv6_dst;            /* destination address */
        string ipv6_saddr;               /* source address, string */
        string ipv6_daddr;               /* destination address, string */
        ip6_t *ipv6_hdr;                 /* pointer to raw header */
} ipv6info_t;
```

The `ipv4info_t` and `ipv6info_t` export fields of the IP headers, after network to host byte order correction. There are also versions of the source and destination addresses, converted to strings, available in these structures, such as `args[2]->ip_saddr`, which performs the translation automatically whether it is IPv4 or IPv6.

Network Providers

The ip provider is the first in a planned series of stable network providers, which includes providers for TCP, UDP, ARP, and ICMP. This project is described on the "Network Providers" page[3] on the OpenSolaris Web site and by the ip, tcp, and udp provider sections in the DTrace Guide. While writing this book, the tcp and udp providers were successfully integrated into Solaris Nevada (build 142),[4] and work on the next providers (sctp, icmp) is underway.

Example One-Liners

Here we count Web server–received packets by client IP address:

```
solaris# dtrace -n 'tcp:::receive /args[4]->tcp_dport == 80/ {
        @pkts[args[2]->ip_daddr] = count();
}'
dtrace: description 'tcp:::receive' matched 1 probe
^C

  192.168.1.8                                                       9
  fe80::214:4fff:fe3b:76c8                                         12
  192.168.1.51                                                     32
  10.1.70.16                                                       83
  192.168.7.3                                                     121
  192.168.101.101                                                 192
```

Here we count established TCP connections by port number:

```
solaris# dtrace -n 'tcp:::accept-established { @[args[4]->tcp_dport] = count(); }'
dtrace: description 'tcp:::accept-established' matched 1 probe
^C

       79                2
       22               14
       80              327
```

3. *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/NetworkProvider*

4. The project identifier is PSARC/2010/106, "DTrace TCP and UDP providers," and was designed and developed by Brendan Gregg and Alan Maguire.

Network Provider Collection

The collection of stable network providers has been designed with the providers and arguments shown in Tables 6-6, 6-7, and 6-8.

Table 6-6 Planned Network Providers

Provider	Description
gld	Traces the generic LAN device layer and shows link layer activity such as Ethernet frames. The probes allow frame-by-frame tracing.
arp	Traces ARP and RARP packets.
icmp	Traces ICMP packets and provides the type and code from the ICMP header.
ip	Traces IP details for IPv4 and IPv6 send and receive I/O.
tcp	Traces the TCP layer, showing send/receive I/O, connections, and state changes.
udp	Traces User Datagram Protocol and send and receive I/O.
sctp	Traces the Stream Control Transmission Protocol.
socket	Traces the socket layer, close to the application. These probes fire in the same context as the corresponding process, and show socket I/O.

Table 6-7 Planned Network Provider Arguments

Probes	args[0]	args[1]	args[2]	args[3]	args[4]	args[5]
gld:::send gld:::receive	pktinfo_t *	NULL	ipinfo_t *	illinfo_t *	etherinfo_t *	
ip:::send ip:::receive	pktinfo_t *	csinfo_t *	ipinfo_t *	illinfo_t *	ipv4info_t *	ipv6info_t *
tcp:::send tcp:::receive	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
tcp:::accept-* tcp:::connect-*	pktinfo_t *	csinfo_t *	ipinfo_t *	tcpsinfo_t *	tcpinfo_t *	
tcp:::state-change	NULL	csinfo_t *	NULL	tcpnsinfo_t *	NULL	tcplsinfo_t *
udp:::send udp:::receive	pktinfo_t *	csinfo_t *	ipinfo_t *	udpinfo_t *		
udp:::stream-*	pktinfo_t *					
sctp:::send sctp:::receive	pktinfo_t *	csinfo_t *	ipinfo_t *	sctpsinfo_t *	sctpinfo_t *	
sctp:::state-change	NULL	csinfo_t *	NULL	sctpsinfo_t *	NULL	sctplsinfo_t *
icmp:::send icmp:::receive	pktinfo_t *	csinfo_t *	ipinfo_t *	NULL	icmpinfo_t *	

Table 6-8 Planned Network Provider Argument Types

Type	Description
pktinfo_t	Packet info: includes packet IDs
csinfo_t	Connection state info: includes connection IDs
ipinfo_t	IP info available throughout the stack: IP protocol version, source and destination address (as a string), payload length
ifinfo_t	Interface info: details about the network interface
etherinfo_t	Ethernet header info
ipv4info_t	IPv4 header info
ipv6info_t	IPv6 header info
tcpinfo_t	TCP header info
tcpsinfo_t	TCP connection state info (new state)
tcplsinfo_t	TCP connection last state info (previous state)
udpinfo_t	UDP header info
sctpinfo_t	SCTP header info
sctpsinfo_t	SCTP connection state info (new state)
sctplsinfo_t	SCTP connection last state info (previous state)
icmpinfo_t	ICMP header info

See the DTrace Guide for full documentation of existing and proposed providers.[5] Some (or all) of these providers may be unavailable on your operating system version; if they are unavailable, treat this as a preview of new providers coming up in DTrace and the enhanced capabilities that they will enable. Until they are available, all networking layers can still be traced using the fbt provider.

Example Scripts

The following scripts further demonstrate the role of the network providers by showing example usage. As with the previous one-liners, these scripts demonstrate the tcp provider. More examples of tcp provider scripts are in the "TCP Scripts" section and in the tcp provider section of the DTrace Guide.[6]

5. *http://wikis.sun.com/display/DTrace*
6. *http://wikis.sun.com/display/DTrace/tcp+Provider*

tcpconnlat.d

TCP connection latency is a very useful metric and can provide insight into the target server load. The following script was designed to measure it from the client:

```
1   #!/usr/sbin/dtrace -s
2
3   tcp:::connect-request
4   {
5           start[args[1]->cs_cid] = timestamp;
6   }
7
8   tcp:::connect-established
9   /start[args[1]->cs_cid]/
10  {
11          @latency["Connect Latency (ns)", args[2]->ip_daddr] =
12              quantize(timestamp - start[args[1]->cs_cid]);
13          start[args[1]->cs_cid] = 0;
14  }
```

Script tcpconnlat.d

The connection request time stamp is saved to an associative array called `start` (line 5), which is keyed on `args[1]->cs_cid`, which is a unique connection identifier for this TCP session. The saved time stamp is retrieved when the connection is established to calculate the connection time. Executing this script yields the following:

```
solaris# tcpconnlat.d
dtrace: script 'tcpconnlat.d' matched 2 probes
^C

  Connect Latency (ns)                               192.168.1.109
           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@                 3
          262144 |@@@@@@@@@@@@@@@@                         2
          524288 |                                         0

  Connect Latency (ns)                               72.5.124.61
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           3
        16777216 |@@@@@@@@@@                               1
        33554432 |                                         0
```

The output shows that the host 72.5.124.61 was slower to establish a TCP connection than host 192.168.1.109.

The `tcpconnlat.d` script is discussed in more detail in the "TCP Scripts" section.

tcpstate.d

This script shows TCP state changes along with delta times. This assumes that only one TCP session is actively changing state. For it to track multiple TCP sessions properly, the time stamp will need to be saved to an associative array keyed on a `csinfo_t` identifier for that session (`arg0` as in `tcpconnlat.d`):

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10

dtrace:::BEGIN
{
        printf(" %3s %12s  %-20s    %-20s\n",
             "CPU", "DELTA(us)", "OLD", "NEW");
        last = timestamp;
}

tcp:::state-change
{
        this->elapsed = (timestamp - last) / 1000;
        printf(" %3d %12d  %-20s -> %-20s\n", cpu, this->elapsed,
            args[5]->tcps_state, args[3]->tcps_state);
        last = timestamp;
}
```

Script tcpstate.d

Using `tcpstate.d` to trace state changes yields the following:

```
solaris# tcpstate.d
 CPU    DELTA(us)  OLD                   NEW
   3       938491  state-syn-received   -> state-syn-received
   3           98  state-syn-received   -> state-established
   3     14052789  state-established    -> state-close-wait
   3           67  state-close-wait     -> state-last-ack
   3           56  state-last-ack       -> state-bound
   2         7783  state-bound          -> state-closed
   2     68797522  state-idle           -> state-bound
   2          172  state-bound          -> state-syn-sent
   3          210  state-syn-sent       -> state-established
   2         5364  state-established    -> state-fin-wait1
   3           79  state-fin-wait1      -> state-fin-wait2
   3           65  state-fin-wait2      -> state-time-wait
```

fbt Provider

The fbt provider can be used to examine *all* the functions in the network stack, the function arguments, the return codes, the return instruction offsets, and both the elapsed time and on-CPU time. See the "fbt Provider" chapter of the DTrace Guide for the full reference.[7]

To navigate this capability for network stack I/O, kernel stack traces may be examined using DTrace as a list of potential fbt probes and their relationships. Each line of the stack trace can be probed individually. Examining stack traces is also a quick way to become familiar with a complex body of code, such as the network stack.

Using the fbt provider should be considered a last resort, as we mentioned in the "Strategy" section. Writing scripts based on the fbt provider ties them to a particular operating system and kernel version, because they instrument the raw kernel source code. When the kernel is upgraded, the fbt scripts may need to be rewritten to follow changes in function names and arguments. The Solaris network stack implementation does change regularly with kernel updates, so you should expect fbt scripts to require maintenance. The `tcpsnoop` script (discussed later) is an example of this, because it was originally written using the fbt provider and broke several times because of kernel upgrades. What's most important to remember is that fbt tracing of these functions is nonetheless *possible* using DTrace, should the need arise.

The following sections show how to trace send and receive packets using the fbt provider to illustrate the capability (and complexity) of fbt-based tracing.

Send

Sending a packet ends with the network device driver send function. By looking at the stack backtrace at this point, we can see the path taken through the kernel to send a packet.

Solaris

A Solaris system with an `nge` network interface was traced; this version of the `nge` driver has a function called `nge_send()`, from which the stack backtrace was counted:

```
solaris# dtrace -n 'fbt::nge_send:entry { @[probefunc, stack()] = count(); }'
dtrace: description 'fbt::nge_send:entry ' matched 1 probe
^C
[...]
  nge_send
              nge`nge_m_tx+0x57
              dls`dls_tx+0x1d
              dld`dld_tx_single+0x2a
              dld`str_mdata_fastpath_put+0x7f
```

7. *http://wikis.sun.com/display/DTrace/fbt+Provider*

```
                ip`tcp_send_data+0x7c9
                ip`tcp_send+0xb1b
                ip`tcp_wput_data+0x75a
                ip`tcp_output+0x7c5
                ip`squeue_enter+0x416
                ip`tcp_wput+0xf8
                sockfs`sostream_direct+0x168
                sockfs`socktpi_write+0x179
                genunix`fop_write+0x69
                genunix`write+0x208
                genunix`write32+0x1e
                unix`sys_syscall32+0x1fc
                721
```

The path from the write syscall at the bottom of the stack through to the network interface at the top can be seen. If you try this yourself on any Solaris version (and trace the send routine of the network driver you are using), there is a very good chance that the network stack trace will look slightly different: Different workloads are processed in different ways, and the network stack implementation changes over time (that's why the fbt provider is an unstable solution).

The same DTrace one-liner and workload was executed on the latest version of the Solaris Nevada[8] kernel, and the stack has indeed changed:

```
solaris# dtrace -n 'fbt::nge_send:entry { @[probefunc, stack()] = count(); }'
dtrace: description 'fbt::nge_send:entry ' matched 1 probe
^C
[...]
  nge_send
                nge`nge_m_tx+0x60
                mac`mac_tx+0x2c4
                dld`str_mdata_fastpath_put+0xa4
                ip`tcp_send_data+0x94e
                ip`tcp_send+0xb69
                ip`tcp_wput_data+0x72c
                ip`tcp_rput_data+0x3342
                ip`squeue_drain+0x17f
                ip`squeue_enter+0x3f4
                ip`tcp_sendmsg+0xfd
                sockfs`so_sendmsg+0x1c7
                sockfs`socket_sendmsg+0x61
                sockfs`socket_vop_write+0x63
                genunix`fop_write+0xa4
                genunix`write+0x2e2
                genunix`write32+0x22
                unix`sys_syscall32+0x101
                766
```

This newer kernel appears to have replaced the `dls_tx()` function with `mac_tx()`. This illustrates that any fbt-based script will need to be modified to match the underlying source it is tracing. The same fbt-based script is unlikely to work on

8. Solaris Nevada is the current development version of Solaris.

different versions of the Solaris kernel, which may implement networking using slightly different sets of functions. For this reason, an fbt-based script that executes correctly on a given server may fail after that server has had a software upgrade or kernel patch applied.

Maintaining fbt-based scripts is a known nuisance. Still, tracing at this level is very valuable in spite of the drawbacks. Any areas commonly traced using fbt should eventually have stable providers available, such as the ip provider for IP and the tcp provider for TCP. Stable scripts won't require such maintenance and should work everywhere that the provider is available.

That said, the previous two examples illustrate the value derived from a relatively simple DTrace invocation. The code path through the kernel from the application-issued system call through to the network driver send function provides a series of instrumentation points that may be useful when investigating or examining network activity.

Mac OS X

On this version of Mac OS X (10.6), functions for the network device driver were not visible to DTrace, which can happen if symbol information is stripped from binaries. Instead of tracing from the device driver, outbound network I/O can be traced as far as the `ether_frameout()` function in the OS X kernel:

```
macosx# dtrace -n 'fbt::ether_frameout:entry { @[probefunc, stack()] = count() }'
dtrace: description 'fbt::ether_frameout:entry ' matched 1 probe
^C

  ether_frameout
              mach_kernel`ifnet_input+0xe43
              mach_kernel`ifnet_output+0x4d
              mach_kernel`ip_output_list+0x1d9f
              mach_kernel`tcp_setpersist+0x16e
              mach_kernel`tcp_output+0x17ab
              mach_kernel`tcp_ctloutput+0x453
              mach_kernel`sosend+0x84e
              mach_kernel`fill_pipeinfo+0x9e0
              mach_kernel`readv+0x138
              mach_kernel`write_nocancel+0xb4
              mach_kernel`unix_syscall+0x23c
              mach_kernel`lo_unix_scall+0xea
               15
[...]
```

We can see the path from the write system call at the bottom of the stack through to `ether_frameout()` at the top. As with the Solaris network stack, these functions may change from software release to release.

FreeBSD

This FreeBSD 8.0 system uses an "em" interface for Ethernet, from which the `em_xmit()` function was traced along with the kernel stack backtrace:

```
freebsd# dtrace -n 'fbt::em_xmit:entry { @[probefunc, stack()] = count(); }'
dtrace: description 'fbt::em_xmit:entry ' matched 1 probe
^C

  em_xmit
              kernel`em_mq_start_locked+0x14f
              kernel`em_mq_start+0x50
              kernel`ether_output_frame+0x60
              kernel`ether_output+0x5de
              kernel`ip_output+0x9ce
              kernel`tcp_output+0x14cf
              kernel`tcp_usr_send+0x28a
              kernel`sosend_generic+0x645
              kernel`sosend+0x3f
              kernel`soo_write+0x63
              kernel`dofilewrite+0x97
              kernel`kern_writev+0x58
              kernel`write+0x4f
              kernel`syscall+0x3e5
              kernel`0xc0bc2030
               38
```

This shows the path from write system call to the em interface. Key TCP/IP stack functions include `tcp_output()`, `ip_output()`, and `ether_output()`.

Some of the functions in the stack have similar names to the Mac OS X stack, which is not surprising: The Mac OS X kernel has components based on the FreeBSD and 4.4BSD code.[9]

Receive

Receiving network I/O usually ends with the application completing a `read()` (or equivalent) system call. However, the kernel stack trace here does not show the full stack when the read return probe is instrumented:

```
# dtrace -n 'syscall::read*:return /execname == "ttcp"/ { @[stack()] = count(); }'
dtrace: description 'syscall::read*:return ' matched 3 probes
^C

              unix`_sys_sysenter_post_swapgs+0x14b
            10001
```

9. Code sharing is not uncommon between operating systems; other instances include the ZFS code from OpenSolaris being ported to both Mac OS X and FreeBSD, and, of course, the DTrace code itself.

This is because the stack shown previously has occurred after a context switch to the application thread, from a kernel thread that performed the TCP/IP processing. Briefly, receive packets are typically processed in the interrupt handler of the NIC driver. Subsequent receive packet handling may be passed off to other functions in the driver code to minimize time spent in interrupt context. The key point here is that tracing the code path through the kernel for network receive packet handling is challenging because of the asynchronous nature of the receive event, interrupt processing, and the subsequent context switching that occurs when the receive data is made available to the application by the kernel.

Solaris

The sched provider can be used to show the stack trace just before application threads are scheduled. To *only* show those stack traces from TCP receives, the mib provider is used to check that the kernel thread did process TCP/IP while executing this thread:

```
solaris# dtrace -n 'mib:::tcpInDataInorderBytes { self->in = 1; } sched:::enqueue
/self->in/ { @[args[1]->pr_fname, stack()] = count(); self->in = 0; }'
dtrace: description 'mib:::tcpInDataInorderBytes ' matched 6 probes
^C
[...]
  ttcp
              TS`ts_wakeup+0x188
              genunix`sleepq_wakeall_chan+0x7c
              genunix`cv_broadcast+0x78
              genunix`strrput+0x56e
              unix`putnext+0x2f1
              ip`tcp_rcv_drain+0xf9
              ip`tcp_rput_data+0x2acf
              ip`squeue_enter_chain+0x2c0
              ip`ip_input+0xa42
              dls`i_dls_link_rx+0x2b9
              mac`mac_do_rx+0xba
              mac`mac_rx+0x1b
              nge`nge_receive+0x47
              nge`nge_intr_handle+0xbd
              nge`nge_chip_intr+0xca
              unix`av_dispatch_autovect+0x8c
              unix`dispatch_hardint+0x2f
              unix`switch_sp_and_call+0x13
             1345
```

At the top of the stack is the `TS ts_wakeup()` function, which is the time share scheduling class-specific code for waking up a sleeping thread. Using the `args[1]->pr_fname` data from the sched provider as an aggregation key (along with `stack()`), we are able to see the ttcp process getting the wake-up and placed on a run queue (`sched:::enqueue`). The stack trace includes key TCP/IP functions such as `ip_input()` and `tcp_rput_data()`.

Mac OS X

Mac OS X does not currently have stable mib and sched providers, so the unstable fbt provider was used to extract similar information:

```
macosx# dtrace -n 'syscall:::entry { name[(uint64_t)curthread] = execname; }
fbt::tcp_input:entry { self->tcp = 1; }
fbt::thread_unblock:entry /self->tcp && name[arg0] != NULL/
{ @[name[arg0], stack()] = count(); }'
dtrace: description 'syscall:::entry ' matched 429 probes
^C

  sshd
              mach_kernel`thread_go+0x2a
              mach_kernel`wait_queue_assert_wait64+0x24b
              mach_kernel`wait_queue_wakeup_all+0x8b
              mach_kernel`selwakeup+0x40
              mach_kernel`sowakeup+0x25
              mach_kernel`sorwakeup+0x23
              mach_kernel`tcp_input+0x1bbb
              mach_kernel`ip_rsvp_done+0x1c6
              mach_kernel`ip_input+0x17bd
              mach_kernel`ip_input+0x17f9
              mach_kernel`proto_input+0x92
              mach_kernel`ether_detach_inet+0x1c9
              mach_kernel`ifnet_input+0x2f8
              mach_kernel`ifnet_input+0xa51
              mach_kernel`ifnet_input+0xcaf
              mach_kernel`call_continuation+0x1c
               10
```

The stack trace includes key TCP/IP functions such as `ip_input()` and `tcp_input()`.

FreeBSD

The stack trace on FreeBSD is derived similarly to Mac OS X (skipping the `execname` in this case):

```
freebsd# dtrace -n 'fbt::tcp_input:entry { self->ok = 1; }
fbt::thread_lock_unblock:entry /self->ok/ { @[stack()] = count(); }'
dtrace: description 'fbt::tcp_input:entry ' matched 2 probes
^C

              kernel`sched_add+0xf2
              kernel`sched_wakeup+0x69
              kernel`setrunnable+0x88
              kernel`sleepq_resume_thread+0xc8
              kernel`sleepq_broadcast+0x8b
              kernel`cv_broadcastpri+0x4d
              kernel`doselwakeup+0xe6
              kernel`selwakeuppri+0xe
              kernel`sowakeup+0x1f
              kernel`tcp_do_segment+0x946
              kernel`tcp_input+0x11c0
              kernel`ip_input+0x6aa
```

continues

```
                kernel`netisr_dispatch_src+0x89
                kernel`netisr_dispatch+0x20
                kernel`ether_demux+0x161
                kernel`ether_input+0x313
                kernel`em_rxeof+0x4fa
                kernel`em_handle_rxtx+0x27
                kernel`taskqueue_run+0x162
                kernel`taskqueue_thread_loop+0xbd
                 30
```

As on Mac OS X, this FreeBSD trace includes key TCP/IP functions such as `ip_input()` and `tcp_input()` (and as mentioned earlier, similarities are not surprising because of similar origins of the kernel code).

One-Liners

The following DTrace one-liners are grouped by provider. Not all providers may be available on your operating system version, especially newer providers such as tcp and udp.

syscall Provider

The following one-liners demonstrate the use of the syscall provider for observing socket and network activity.

Socket accepts by process name:

```
dtrace -n 'syscall::accept*:entry { @[execname] = count(); }'
```

Socket connections by process and user stack trace:

```
dtrace -n 'syscall::connect*:entry { trace(execname); ustack(); }'
```

Socket read, write, send, recv I/O count by syscall:

```
dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/ { @[probefunc]
= count(); }'
```

```
dtrace -n 'syscall::write*:entry /fds[arg0].fi_fs == "sockfs"/ { @[probefunc]
= count();}'
```

```
dtrace -n 'syscall::send*:entry /fds[arg0].fi_fs == "sockfs"/ { @[probefunc]
= count(); }'
```

```
dtrace -n 'syscall::recv*:entry /fds[arg0].fi_fs == "sockfs"/ { @[probefunc]
= count(); }'
```

Socket read (write/send/recv) I/O count by process name:

```
dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/ { @[execname]
= count(); }'
```

Socket reads (write/send/recv) I/O count by syscall and process name:

```
dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/
{ @[strjoin(probefunc, strjoin("() by ", execname))] = count(); }'
```

Socket reads (write/send/recv) I/O count by process and user stack trace:

```
dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/ { @[execname, ustack()]
= count(); }'
```

Socket write requested bytes by process name:

```
dtrace -n 'syscall::write:entry /fds[arg0].fi_fs == "sockfs"/ { @[execname]
= sum(arg2); }'
```

Socket read returned bytes by process name:

```
dtrace -n 'syscall::read:entry /fds[arg0].fi_fs == "sockfs"/ { self->ok = 1; }
syscall::read:return /self->ok/ { @[execname] = sum(arg0); self->ok = 0; }'
```

Socket write requested I/O size distribution by process name:

```
dtrace -n 'syscall::write:entry,syscall::send:entry /fds[arg0].fi_fs == "sockfs"/
{ @[execname] = quantize(arg2); }'
```

mib Provider

The following one-liners demonstrate the use of the mib provider for tracking network events systemwide.

SNMP MIB event count:

```
dtrace -n 'mib::: { @[probename] = count(); }'
```

IP event statistics:

```
dtrace -n 'mib:::ip* { @[probename] = sum(arg0); }'
```

IP event statistics with kernel function:

```
dtrace -n 'mib:::ip* { @[strjoin(probefunc, strjoin("() -> ", probename))]
= sum(arg0); }'
```

TCP event statistics:

```
dtrace -n 'mib:::tcp* /(int)arg0 > 0/ { @[probename] = sum(arg0); }'
```

TCP event statistics with kernel function:

```
dtrace -n 'mib:::tcp* { @[strjoin(probefunc, strjoin("() -> ", probename))]
= sum(arg0);}'
```

UDP event statistics:

```
dtrace -n 'mib:::udp* { @[probename] = sum(arg0); }'
```

ICMP event trace:

```
dtrace -Fn 'mib:::icmp*  { trace(timestamp); }'
dtrace -Fn 'mib::icmp_*: { trace(timestamp); }'
```

ICMP event by kernel stack trace:

```
dtrace -n 'mib:::icmp*  { stack(); }'
dtrace -n 'mib::icmp_*: { stack(); }'
```

ip Provider

The ip provider greatly enhances observing network activity, as shown in the following one-liners.

Received IP packets by host address:

```
dtrace -n 'ip:::receive { @[args[2]->ip_saddr] = count(); }'
```

IP send payload size distribution by destination:

```
dtrace -n 'ip:::send { @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
```

tcp Provider

Variants are demonstrated where similar information can be fetched from different `args[]` locations (see the one-liner examples for more discussion about this).

Watch inbound TCP connections by remote address (either):

```
dtrace -n 'tcp:::accept-established { trace(args[2]->ip_saddr); }'
dtrace -n 'tcp:::accept-established { trace(args[3]->tcps_raddr); }'
```

Inbound TCP connections by remote address summary:

```
dtrace -n 'tcp:::accept-established { @addr[args[3]->tcps_raddr] = count(); }'
```

Inbound TCP connections by local port summary:

```
dtrace -n 'tcp:::accept-established { @port[args[3]->tcps_lport] = count(); }'
```

Who is connecting to what:

```
dtrace -n 'tcp:::accept-established { @[args[3]->tcps_raddr, args[3]->tcps_lport] =
count(); }'
```

Who isn't connecting to what:

```
dtrace -n 'tcp:::accept-refused { @[args[2]->ip_daddr, args[4]->tcp_sport] = count(); }'
```

What am I connecting to?

```
dtrace -n 'tcp:::connect-established { @[args[3]->tcps_raddr , args[3]->tcps_rport] =
count(); }'
```

Outbound TCP connections by remote port summary:

```
dtrace -n 'tcp:::connect-established { @port[args[3]->tcps_rport] = count(); }'
```

TCP received packets by remote address summary (either):

```
dtrace -n 'tcp:::receive { @addr[args[2]->ip_saddr] = count(); }'
dtrace -n 'tcp:::receive { @addr[args[3]->tcps_raddr] = count(); }'
```

TCP sent packets by remote address summary (either):

```
dtrace -n 'tcp:::send { @addr[args[2]->ip_daddr] = count(); }'
dtrace -n 'tcp:::send { @addr[args[3]->tcps_raddr] = count(); }'
```

TCP received packets by local port summary:

```
dtrace -n 'tcp:::receive { @port[args[4]->tcp_dport] = count(); }'
```

TCP send packets by remote port summary:

```
dtrace -n 'tcp:::send { @port[args[4]->tcp_dport] = count(); }'
```

IP payload bytes for TCP send, size distribution by destination address:

```
dtrace -n 'tcp:::send { @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
```

TCP payload bytes for TCP send:

```
dtrace -n 'tcp:::send { @bytes = sum(args[2]->ip_plength - args[4]->tcp_offset); }'
```

TCP events by type summary:

```
dtrace -n 'tcp::: { @[probename] = count(); }'
```

udp Provider

The following one-liners demonstrate the use of the udp provider.

UDP received packets by remote address summary (either):

```
dtrace -n 'udp:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace -n 'udp:::receive { @[args[3]->udps_raddr] = count(); }'
```

UDP sent packets by remote port summary:

```
dtrace -n 'udp:::send { @[args[4]->udp_dport] = count(); }'
```

syscall Provider Examples

In this section, we provide some examples of using the syscall provider in DTrace one-liners to observe network load and activity.

Socket Accepts by Process Name

By tracing the process name for the `accept()` system call, it is possible to identify which processes are accepting socket connections:

```
solaris# dtrace -n 'syscall::accept*:entry { @[execname] = count(); }'
dtrace: description 'syscall::accept*:entry ' matched 1 probe
^C

  sshd                                                              1
  inetd                                                             2
  httpd                                                            15
```

During this one-liner, the `httpd` processes called `accept()` 15 times, which is likely in response to 15 inbound HTTP connections. These `accept()` calls may have actually failed; to check for this, examine the return value and `errno` on the `accept*:return` probes.

Socket Connections by Process and User Stack Trace

It can be useful to know why applications are establishing socket connections. This can be shown with a one-liner to print the process name and user stack trace for the `connect()` system call, which was run on a Solaris client:

```
solaris# dtrace -n 'syscall::connect*:entry { trace(execname); ustack(); }'
dtrace: description 'syscall::connect:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  96749                    connect:entry   ssh
              libc.so.1`_so_connect+0x7
              ssh`timeout_connect+0x151
              ssh`ssh_connect+0x182
              ssh`main+0x928
              ssh`_start+0x7a

  1  96749                    connect:entry   firefox-bin
              libc.so.1`_so_connect+0x7
              libnspr4.so`pt_Connect+0x13c
              libnspr4.so`PR_Connect+0x18
              libnecko.so`__1cRnsSocketTransportOInitiateSocket6M_I_+0x271
              libnecko.so`__1cNnsSocketEventLHandleEvent6FpnHPLEvent__pv_+0x2ce
              libxpcom_core.so`PL_HandleEvent+0x22
              libnecko.so`__1cYnsSocketTransportServiceNServiceEventQdD6M_i_+0x99
              libnecko.so`__1cYnsSocketTransportServiceDRun6M_I_+0x9b0
              libxpcom_core.so`__1cInsThreadEMain6Fpv_v_+0x74
              libnspr4.so`_pt_root+0xd1
              libc.so.1`_thr_setup+0x52
              libc.so.1`_lwp_start
[...]
```

This shows the user stack traces for processes named `ssh` (SSH client) and `firefox-bin` (Mozilla Firefox Web browser), because they established connections. The stack traces may shed light on why applications are performing socket connections (or, they may be inscrutable without access to source code to follow).

Executing the same one-liner on Mac OS X yields the following:

```
macosx# dtrace -n 'syscall::connect*:entry { trace(execname); ustack(); }'
dtrace: description 'syscall::connect*:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  17914                    connect:entry   ssh
              libSystem.B.dylib`connect$UNIX2003+0xa
              ssh`0x32f99e59
              ssh`0x32f985ce
              0x2

  1  18536           connect_nocancel:entry   firefox-bin
              libSystem.B.dylib`connect$NOCANCEL$UNIX2003+0xa
              libnspr4.dylib`PR_GetSpecialFD+0x85d
              libnspr4.dylib`PR_Connect+0x1f
              XUL`XRE_GetFileFromPath+0x5c97f
              XUL`XRE_GetFileFromPath+0x5dba4
              XUL`std::vector<unsigned short, std::allocator<unsigned short> >::_M_fil
l_insert(__gnu_cxx::__normal_iterator<unsigned short*, >std::vector<unsigned short, st
d::allocator<unsigned short> > >, unsigned long, >unsigned short const&)+0x5129
              XUL`NS_GetComponentRegistrar_P+0x6f73
```

```
                XUL`GetSecurityContext(JNIEnv_*, nsISecurityContext**)+0x2f91d
                XUL`XRE_GetFileFromPath+0x5faea
                XUL`NS_GetComponentRegistrar_P+0x6f73
                XUL`GetSecurityContext(JNIEnv_*, nsISecurityContext**)+0x2f91d
                XUL`NS_GetComponentRegistrar_P+0x71eb
                libnspr4.dylib`PR_Select+0x32c
                libSystem.B.dylib`_pthread_start+0x141
                libSystem.B.dylib`thread_start+0x22
[...]
```

The ssh stack trace shows hexadecimal addresses. These are shown if symbols cannot be translated for some reason, such as the symbol information not being available, or the process exited before DTrace could perform the translation (which is done as a postprocessing step, just before the aggregation is printed).

Socket Read, Write, Send, Recv I/O Count by System Call

The type of socket I/O can be determined by checking which system calls are using socket file descriptors. Here's an example on Solaris:

```
solaris# dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/
 { @[probefunc] = count(); }'
dtrace: description 'syscall::read*:entry ' matched 3 probes
^C

  readv                                                              2
  read                                                            2450
```

They were mostly `read()` system calls to sockets, with a couple of `readv()` system calls. Similar one-liners can be used to investigate writes, sends, and receives:

```
# dtrace -n 'syscall::write*:entry /fds[arg0].fi_fs == "sockfs"/
{ @[probefunc] = count(); }'
# dtrace -n 'syscall::send*:entry /fds[arg0].fi_fs == "sockfs"/
{ @[probefunc] = count(); }'
# dtrace -n 'syscall::recv*:entry /fds[arg0].fi_fs == "sockfs"/
{ @[probefunc] = count(); }'
```

Using fds[] to Identify Socket I/O

The fds (file descriptors) array was a feature added to DTrace after the initial release. This allows the following predicates to be used to match socket I/O:

Solaris: `/fds[arg0].fi_fs == "sockfs"/`

Mac OS X: `/fds[arg0].fi_name == "<socket>"/`

FreeBSD: (`fds[]` array not yet supported)

The previous one-liners for socket I/O used the Solaris predicate; change to the Mac OS X predicate if desired. Some of the scripts, such as `socketio.d`, test for both so that the same script executes on both operating systems.

If you are using an early version of DTrace on Solaris 10 that doesn't have the fds array, you may be able to add it by writing scripts and copying the fds translator to the top of your script (or upgrade to a newer version of Solaris):

```
inline fileinfo_t fds[int fd] = xlate <fileinfo_t> (
    fd >= 0 && fd < curthread->t_procp->p_user.u_finfo.fi_nfiles ?
    curthread->t_procp->p_user.u_finfo.fi_list[fd].uf_file : NULL);
```

For FreeBSD, you will need to dig this information out of the kernel, which won't be easy or stable (but it should be possible!), until `fds[]` is supported.

Socket Read (Write/Send/Recv) I/O Count by Process Name

Since the `syscall` probes fire in process context, socket I/O types can be identified by process by aggregating on `execname`:

```
solaris# dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/
 { @[execname] = count(); }'
dtrace: description 'syscall::read*:entry ' matched 3 probes
^C

  FvwmButtons                                                          2
  FvwmIconMan                                                          2
  finger                                                               2
  xbiff                                                                2
  xclock                                                               2
  xload                                                                8
  gconfd-2                                                            10
  opera                                                               16
  firefox-bin                                                         44
  soffice.bin                                                         89
  ssh                                                                 94
  FvwmPager                                                          123
  gnome-terminal                                                     762
  fvwm2                                                             1898
  realplay.bin                                                      2493
  Xorg                                                              3785
```

Here Xorg (a window manager) called the most socket reads, which we would expect to be localhost I/O. This one-liner can be customized to trace other socket I/O types: write/send/receive.

Socket Reads (Write/Send/Recv) I/O Count by System Call and Process Name

Counting both socket I/O type and process name in the same one-liner yields the following:

```
solaris# dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/
 { @[strjoin(probefunc, strjoin("() by ", execname))] = count(); }'
dtrace: description 'syscall::read*:entry ' matched 3 probes
^C

  read() by xbiff                                                    1
  read() by xclock                                                   1
  read() by FvwmIconMan                                              2
  read() by java                                                     2
  read() by FvwmButtons                                              4
  read() by xterm                                                    4
  readv() by soffice.bin                                             4
  read() by xload                                                    6
  read() by FvwmAnimate                                              8
  readv() by opera                                                  16
  readv() by fvwm2                                                  18
  read() by pidgin                                                  26
  read() by firefox-bin                                             92
  read() by soffice.bin                                            122
  read() by ssh                                                    132
  read() by FvwmPager                                              137
  read() by gnome-terminal                                         310
  read() by realplay.bin                                          1507
  read() by fvwm2                                                 1933
  read() by opera                                                19309
  read() by Xorg                                                 22396
```

The use of `strjoin()` in this one-liner is to make the output cleaner by keeping the string elements together in one key of the aggregation.

Socket Reads (Write/Send/Recv) I/O Count by Process and User Stack Trace

User stack traces can show why I/O was performed, by showing the user level functions which led to that I/O. This one-liner frequency counts the process name and the user stack trace, in this case for reads:

```
solaris# dtrace -n 'syscall::read*:entry /fds[arg0].fi_fs == "sockfs"/
 { @[execname, ustack()] = count(); }'
dtrace: description 'syscall::read*:entry ' matched 3 probes
^C
[...]
  firefox-bin
              libc.so.1`_read+0x7
              libnspr4.so`pt_SocketRead+0x5d
              libnspr4.so`PR_Read+0x18
              libnecko.so`__1cTnsSocketInputStreamERead6MpcIpI_I_+0xf8
              libnecko.so`__1cQnsHttpConnectionOOnWriteSegment6MpcIpI_I_+0x38
              libnecko.so`__1cRnsHttpTransactionQWritePipeSegment6FpnPnsIOutputStream_p
vpcIIpI_I_+0x48
              libxpcom_core.so`__1cSnsPipeOutputStreamNWriteSegments6MpFpnPnsIOutputStr
eam_pvpcIIpI_I3I5_I_+0x309
             libnecko.so`__1cRnsHttpTransactionNWriteSegments6MpnUnsAHttpSegmentWriter
_IpI_I_+0x61
              libnecko.so`__1cQnsHttpConnectionSOnInputStreamReady6MpnTnsIAsyncInputStr
eam__I_+0xc8
              libnecko.so`__1cTnsSocketInputStreamNOnSocketReady6MI_v_+0xca
              libnecko.so`__1cRnsSocketTransportNOnSocketReady6MpnKPRFileDesc_h_v_+0xe5
```
continues

```
              libnecko.so`__1cYnsSocketTransportServiceDRun6M_I_+0x742
              libxpcom_core.so`__1cInsThreadEMain6Fpv_v_+0x74
              libnspr4.so`_pt_root+0xd1
              libc.so.1`_thr_setup+0x52
              libc.so.1`_lwp_start
               28
[...]

  Xorg
              libc.so.1`_read+0xa
              Xorg`_XSERVTransSocketRead+0xf
              Xorg`ReadRequestFromClient+0x14a
              Xorg`Dispatch+0x2fa
              Xorg`main+0x495
              Xorg`_start+0x6c
              984
```

The previous `firefox-bin` stack shows some C++ signatures (mangled function names). These can be postprocessed by c++filt or gc++filt, revealing the human-readable form:

```
firefox-bin
              libc.so.1`_read+0x7
              libnspr4.so`pt_SocketRead+0x5d
              libnspr4.so`PR_Read+0x18
              libnecko.so`unsigned nsSocketInputStream::Read(char*,unsigned,unsigned*)+
0xf8
              libnecko.so`unsigned nsHttpConnection::OnWriteSegment(char*,unsigned,unsi
gned*)+0x38
              libnecko.so`unsigned nsHttpTransaction::WritePipeSegment(nsIOutputStream*
,void*,char*,unsigned,unsigned,unsigned*)+0x48
              libxpcom_core.so`unsigned nsPipeOutputStream::WriteSegments(unsigned(*)(n
sIOutputStream*,void*,char*,unsigned,unsigned,unsigned*),void*,unsigned,unsigned*)+0x3
09
              libnecko.so`unsigned nsHttpTransaction::WriteSegments(nsAHttpSegmentWrite
r*,unsigned,unsigned*)+0x61
              libnecko.so`unsigned nsHttpConnection::OnInputStreamReady(nsIAsyncInputSt
ream*)+0xc8
              libnecko.so`void nsSocketInputStream::OnSocketReady(unsigned)+0xca
              libnecko.so`void nsSocketTransport::OnSocketReady(PRFileDesc*,short)+0xe5
              libnecko.so`unsigned nsSocketTransportService::Run()+0x742
              libxpcom_core.so`void nsThread::Main(void*)+0x74
              libnspr4.so`_pt_root+0xd1
              libc.so.1`_thr_setup+0x52
              libc.so.1`_lwp_start
               28
```

This stack shows that `firefox-bin` was performing socket I/O for the `nsHttpTransaction` module, most probably to fetch Web sites over the HTTP protocol.

Socket Write Bytes by Process Name

The number of bytes of socket I/O can be examined to identify I/O throughput. Here the `write()` system call on sockets is traced, to show the total number of bytes written, by process name:

```
solaris# dtrace -n 'syscall::write:entry /fds[arg0].fi_fs == "sockfs"/
 { @[execname] = sum(arg2); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

  xload                                                              100
  FvwmButtons                                                       1172
  FvwmAnimate                                                       1856
  FvwmPager                                                         4048
  FvwmIconMan                                                       6376
  java                                                              6556
  realplay.bin                                                     17540
  gnome-terminal                                                   18192
  fvwm2                                                            31436
  xclock                                                           71900
  soffice.bin                                                      90364
```

Many of these bytes may be for loopback socket connections; with further analysis, DTrace can tell us whether this is the case.

Socket Read Bytes by Process Name

When tracing socket reads by size, the file descriptor needed to identify socket I/O is available on `read:entry`, while the number of bytes is available on `read:return`. Both must be probed, and associated, to trace socket read bytes. Here we show totals by process name:

```
solaris# dtrace -n 'syscall::read:entry /fds[arg0].fi_fs == "sockfs"/
{ self->ok = 1; } syscall::read:return /self->ok/
{ @[execname] = sum(arg0); self->ok = 0; }'
dtrace: description 'syscall::read:entry ' matched 2 probes
^C

  FvwmAnimate                                                        124
  xload                                                              128
  soffice.bin                                                        288
  opera                                                              384
  FvwmPager                                                         1231
  ssh                                                               1312
  gnome-terminal                                                    5236
  fvwm2                                                            22206
  realplay.bin                                                    360157
  firefox-bin                                                    1049057
  Xorg                                                           1097685
```

The output shows the `firefox-bin` application read over 1MB over sockets, using `read()`, during the sampling period.

Socket Write I/O Size Distribution by Process Name

To better understand socket I/O counts and sizes, distribution plots can be traced for I/O size. Here plots are generated for socket `write()` and `send()` system calls, by process name:

```
# dtrace -n 'syscall::write:entry,syscall::send:entry
/fds[arg0].fi_fs == "sockfs"/ { @[execname] = quantize(arg2); }'
dtrace: description 'syscall::write:entry,syscall::send:entry ' matched 2 probes
^C
[...]

  ssh
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 44
              64 |                                         0

[...]

  firefox-bin
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@                                  9
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             34
            1024 |@@@@                                     5
            2048 |@                                        1
            4096 |                                         0
```

This output shows that the ssh command was writing in sizes of 32 to 63 bytes; such a small size is expected because ssh will be sending encrypted keystrokes. The Firefox browser is sending at least 256 bytes per socket write; for HTTP requests, this will include the HTTP header.

mib Provider Examples

In this section, we demonstrate use of the mib provider.

SNMP MIB Event Count

To get an idea of the various SNMP MIB probes available, they were frequency counted on a host while various network I/O was occurring, including outbound ICMP:

```
solaris# dtrace -n 'mib::: { @[probename] = count(); }'
dtrace: description 'mib::: ' matched 568 probes
^C

  icmpInEchoReps                                                    1
  icmpInMsgs                                                        1
  rawipInDatagrams                                                  1
  rawipOutDatagrams                                                 1
  tcpInDataDupBytes                                                 1
  tcpInDataDupSegs                                                  1
  tcpInDupAck                                                       1
  udpIfStatsNoPorts                                                 1
  tcpActiveOpens                                                    2
[...]
  tcpOutDataBytes                                                1218
  tcpOutDataSegs                                                 1218
```

```
  tcpInDataInorderBytes                                        1336
  tcpInDataInorderSegs                                         1336
  ipIfStatsHCInDelivers                                        1454
  ipIfStatsHCInOctets                                          1462
  ipIfStatsHCInReceives                                        1462
  ipIfStatsHCOutOctets                                         1548
  ipIfStatsHCOutRequests                                       1548
  ipIfStatsHCOutTransmits                                      1548
```

The most frequent events are the `ipIfStats*` events, tracing IP stack events to the network interface I/O. The icmp events can be seen at the top of the output, firing once each.

IP Event Statistics

This one-liner assumes that ip statistics from the mib provider begin with the letters `ip`. If this was not entirely accurate, a script could be written to individually name all the correct ip statistics from the mib provider.

This shows IP statistics from the mib provider when receiving 10,240,000 TCP bytes from a known network test load:

```
solaris# dtrace -n 'mib:::ip* { @[probename] = sum(arg0); }'
dtrace: description 'mib:::ip* ' matched 334 probes
^C

  ipIfStatsForwProhibits                                          1
  ipIfStatsHCInBcastPkts                                          4
  ipIfStatsHCOutRequests                                       1182
  ipIfStatsHCOutTransmits                                      1182
  ipIfStatsHCInDelivers                                        7413
  ipIfStatsHCInReceives                                        7418
  ipIfStatsHCOutOctets                                        48588
  ipIfStatsHCInOctets                                      10597089
```

The byte count is shown in the `ipIfStatsHCInOctets`: IP Interface Statistics High Capacity (64-bit) Inbound Octets (bytes) metric. The actual value of 10,597,089 is slightly larger than the 10,240,000 bytes sent because it includes network overhead, such as packet headers, ACK packets, and so on.

For sent TCP bytes, the `ipIfStatsHCOutOctets` counter will be incremented.

IP Event Statistics with Kernel Function

Similar to the previous one-liner, this includes the (unstable) `probefunc` member of the probe name, which shows the function in the kernel that caused the mib probe to fire:

```
solaris# dtrace -n 'mib:::ip*
{ @[strjoin(probefunc, strjoin("() -> ", probename))] = sum(arg0); }'
dtrace: description 'mib:::ip* ' matched 334 probes
^C

  ip_input() -> ipIfStatsHCInReceives                                 5040
  ip_tcp_input() -> ipIfStatsHCInDelivers                             5040
  tcp_send_data() -> ipIfStatsHCOutRequests                          10028
  tcp_send_data() -> ipIfStatsHCOutTransmits                         10028
  ip_input() -> ipIfStatsHCInOctets                                 231531
  tcp_send_data() -> ipIfStatsHCOutOctets                         10641374
```

This suggests that `tcp_send_data()` is likely to be a key function for sending IP outbound traffic, because it will cause the `ipIfStatsHCOutOctets` probe to fire.

TCP Event Statistics

Some of the mib TCP events can return negative values in `arg0`,[10] which, if treated as an unsigned value, can be a very large number, producing confusing results. This is checked in a predicate:

```
solaris# dtrace -n 'mib:::tcp* /(int)arg0 > 0/ { @[probename] = sum(arg0); }'
dtrace: description 'mib:::tcp* ' matched 94 probes
^C

  tcpActiveOpens                                                         1
  tcpInDupAck                                                            1
  tcpTimRetrans                                                          1
  tcpOutControl                                                          2
  tcpOutAckDelayed                                                      11
  tcpOutAck                                                             24
  tcpInDataInorderSegs                                                  33
  tcpInAckSegs                                                        5003
  tcpRttUpdate                                                        5003
  tcpOutDataSegs                                                     10002
  tcpInDataInorderBytes                                              24851
  tcpInAckBytes                                                   10240157
  tcpOutDataBytes                                                 14411804
```

TCP Event Statistics with Kernel Function

In the following example, we frequency count kernel functions updating TCP mib statistics:

10. `args[0]` is supposed to be used with the mib provider; however, the tcp* wildcard matches some probes where `args[0]` isn't available, and so DTrace won't allow `args[0]` to be used. Using `arg0` is a workaround.

```
solaris# dtrace -n 'mib:::tcp* /(int)arg0 > 0/
{ @[strjoin(probefunc, strjoin("() -> ", probename))] = sum(arg0); }'
dtrace: description 'mib:::tcp* ' matched 94 probes
^C

  tcp_output() -> tcpOutDataSegs                                      1
  tcp_rput_data() -> tcpInAckSegs                                     1
  tcp_set_rto() -> tcpRttUpdate                                       1
  tcp_ack_timer() -> tcpOutAck                                       10
  tcp_ack_timer() -> tcpOutAckDelayed                                10
  tcp_rput_data() -> tcpOutAck                                       11
  tcp_rput_data() -> tcpInDataInorderSegs                            33
  tcp_output() -> tcpOutDataBytes                                   108
  tcp_rput_data() -> tcpInAckBytes                                  108
  tcp_rput_data() -> tcpInDataInorderBytes                        36847
```

UDP Event Statistics

UDP probes are also available in the mib provider:

```
solaris# dtrace -n 'mib:::udp* { @[probename] = sum(arg0); }'
dtrace: description 'mib:::udp* ' matched 20 probes
^C

  udpIfStatsNoPorts                                                   2
  udpHCOutDatagrams                                                  46
  udpHCInDatagrams                                                   50
```

ICMP Event Trace

Because ICMP events are expected to be less frequent, the ICMP one-liners will trace and print output per event, rather than summarize using aggregations. Either of these will be used:

```
dtrace -Fn 'mib:::icmp*  { trace(timestamp); }'

dtrace -Fn 'mib::icmp_*: { trace(timestamp); }'
```

While the stable method for matching the ICMP probes is to use a wildcard in the probename field (first one-liner), this doesn't match raw IP packets used for outbound ICMP, because their names start with `rawip`.[11] The second one-liner is a workaround, matching all mib probes that fire in `icmp_*` functions:

11. This seems like a bug.

Here we have an inbound ICMP echo request:

```
solaris# dtrace -Fn 'mib::icmp_*: { trace(timestamp); }'
dtrace: description 'mib::icmp_*: ' matched 89 probes
CPU FUNCTION
  1 | icmp_inbound:icmpInMsgs                    5356002727524698
  1 | icmp_inbound:icmpInEchos                   5356002727527913
  1 | icmp_inbound:icmpOutEchoReps               5356002727529442
  1 | icmp_inbound:icmpOutMsgs                   5356002727537901
^C
```

Time stamps are printed to measure latency and to check that the output is in the correct order (multi-CPU systems may require post sorting). Flow indent was used (the `-F` flag), in case you need to customize this one-liner by adding fbt probes for the functions shown. Here's an example:

```
solaris# dtrace -Fn 'mib::icmp_*:,fbt::icmp_inbound: { trace(timestamp); }'
dtrace: description 'mib::icmp_*:,fbt::icmp_inbound: ' matched 91 probes
CPU FUNCTION
  1  -> icmp_inbound                           5356036952771520
  1    | icmp_inbound:icmpInMsgs               5356036952774471
  1    | icmp_inbound:icmpInEchos              5356036952776885
  1    | icmp_inbound:icmpOutEchoReps          5356036952778495
  1    | icmp_inbound:icmpOutMsgs              5356036952786695
  1  <- icmp_inbound                           5356036952807373
^C
```

This suggests that all four mib probes fired in the duration of `icmp_inbound()`. Further DTracing can confirm.

Here's an outbound ICMP echo request:

```
solaris# dtrace -Fn 'mib::icmp_*: { trace(timestamp); }'
dtrace: description 'mib::icmp_*: ' matched 89 probes
CPU FUNCTION
  1 | icmp_wput:rawipOutDatagrams              5356062980086596
  1 | icmp_inbound:icmpInMsgs                  5356062980226177
  1 | icmp_inbound:icmpInEchoReps              5356062980227969
  1 | icmp_input:rawipInDatagrams              5356062980232747
^C
```

Probing the identified kernel functions from the previous output yields the following:

```
solaris# dtrace -Fn'mib::icmp_*:,fbt::icmp_wput:,fbt::icmp_inbound:
,fbt::icmp_input: { trace(timestamp); }'
```

```
dtrace: description 'mib::icmp_*:,fbt::icmp_wput:,fbt::icmp_inbound:,fbt::icmp_input:
' matched 95 probes
CPU FUNCTION
  1  -> icmp_wput                                5356158150344413
  1   | icmp_wput:rawipOutDatagrams              5356158150348528
  1  <- icmp_wput                                5356158150368629
  1  -> icmp_inbound                             5356158150552807
  1   | icmp_inbound:icmpInMsgs                  5356158150554112
  1   | icmp_inbound:icmpInEchoReps              5356158150555917
  1    -> icmp_input                             5356158150560871
  1     | icmp_input:rawipInDatagrams            5356158150563004
  1    <- icmp_input                             5356158150567422
  1  <- icmp_inbound                             5356158150569275
```

Here again, we can use the mib provider to correlate specific kernel functions to network events of interest and use that information to build the next set of DTrace scripts for further analysis.

ICMP Event by Kernel Stack Trace

In addition to identifying the kernel functions that contain the MIB probes, we can print a kernel stack trace and observe the entire code path to the MIB probe:

```
solaris# dtrace -n 'mib::icmp_*: { stack(); }'
dtrace: description 'mib::icmp_*: ' matched 89 probes
CPU     ID                    FUNCTION:NAME
  1  25849      icmp_wput:rawipOutDatagrams
              unix`putnext+0x2f1
              genunix`strput+0x1cf
              genunix`kstrputmsg+0x2bf
              sockfs`sosend_dgram+0x2dd
              sockfs`sotpi_sendmsg+0x566
              sockfs`sendit+0x1b8
              sockfs`sendto+0xb8
              sockfs`sendto32+0x2d
              unix`sys_syscall32+0x1fc

  1  25612          icmp_inbound:icmpInMsgs
              ip`ip_proto_input+0x620
              ip`ip_input+0x9df
              dls`i_dls_link_rx+0x2b9
              mac`mac_do_rx+0xba
              mac`mac_rx+0x1b
              nge`nge_receive+0x47
              nge`nge_intr_handle+0xbd
              nge`nge_chip_intr+0xca
              unix`av_dispatch_autovect+0x8c
              unix`dispatch_hardint+0x2f
              unix`switch_sp_and_call+0x13
[...]
```

In the bottom stack frame, we see the `icmp_inbound()` kernel function causing the `icmpInMsgs` probe to fire, and the path through the kernel on this inbound traffic originated in the network interface (nge) interrupt handler. The top stack frame shows a send over ICMP.

ip Provider Examples

In this section, we demonstrate use of the ip provider.

Received IP Packets by Host Address

If the IP provider is present, summarizing received IP packets by host address is a simple one-liner:

```
# dtrace -n 'ip:::receive { @[args[2]->ip_saddr] = count(); }'
dtrace: description 'ip:::receive ' matched 4 probes
^C

  192.168.1.5                                                       1
  192.168.1.185                                                     4
  fe80::214:4fff:fe3b:76c8                                          9
  127.0.0.1                                                        14
  192.168.1.109                                                    28
```

This includes IPv4 and IPv6 hosts.

IP Send Payload Size Distribution by Destination

The send payload size is shown here by destination host. This may be useful to detect hosts that are supposed to be using jumbo frames but are not:

```
solaris# dtrace -n 'ip:::send
{ @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
dtrace: description 'ip:::send ' matched 11 probes
^C

  192.168.2.27
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          7
              32 |@@@@                                     1
              64 |@@@@                                     1
             128 |                                         0

  192.168.1.109
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |@@@@@                                    5
              32 |@@@                                      3
              64 |@@@@@@@@@@@@@@@@@@@@@@@@@@               24
             128 |@                                        1
             256 |@                                        1
             512 |@@                                       2
            1024 |@                                        1
            2048 |                                         0
```

The output shows that most of the packets sent to the 192.168.1.109 host were in the 64-byte to 127-byte range.

tcp Provider Examples

In this section, we show examples of using the tcp provider.

Inbound TCP Connections by Remote Address Summary

The `accept-established` probe traces passive TCP connection–established events, which are typically for inbound connections to a server. This one-liner summarizes which remote hosts are establishing TCP connections:

```
solaris# dtrace -n 'tcp:::accept-established
{ @addr[args[3]->tcps_raddr] = count(); }'
dtrace: description 'tcp:::state-change' matched 1 probes
^C

  127.0.0.1                                                         1
  192.168.2.88                                                      1
  fe80::214:4fff:fe8d:59aa                                          1
  192.168.1.109                                                     3
```

Since the `accept-established` probe fires in the context of the final ACK in the TCP handshake, the source address in the IP header may also be used to refer to the remote host, so this one-liner can be written as follows:

```
solaris# dtrace -n 'tcp:::accept-established { @addr[args[2]->ip_saddr] = count(); }'
```

The development documentation for the TCP provider uses this approach for writing one-liners, simply because `tcps_raddr` and `tcps_laddr` were not available in `args[3]` until later in the provider development.

Inbound TCP Connections by Local Port Summary

Tracing the local TCP port for connections will show which local services are accepting connections:

```
solaris# dtrace -n 'tcp:::accept-established { @[args[3]->tcps_lport] = count(); }'
dtrace: description 'tcp:::accept-established' matched 1 probes
^C

    22                 1
    80                 7
```

While this one-liner was running, there was one connection to port 22 (SSH) and seven connections to port 80 (HTTP).

Who Is Connecting to What

Combining the previous two one-liners will count which remote hosts are connecting to which local ports:

```
solaris# dtrace -n 'tcp:::accept-established
{ @[args[3]->tcps_raddr, args[3]->tcps_lport] = count(); }'
dtrace: description 'tcp:::accept-established' matched 1 probes
^C

  192.168.2.88                                       40648              1
  fe80::214:4fff:fe8d:59aa                              22              1
  192.168.1.109                                         22              3
```

During tracing, 192.168.1.109 established three connections to local port 22 (SSH).

Who Isn't Connecting to What

As well as tracing successful connections, tracing *un*successful connections can be extremely valuable when troubleshooting network issues.

```
solaris# dtrace -n 'tcp:::accept-refused
{ @[args[3]->tcps_raddr, args[3]->tcps_lport] = count(); }'
dtrace: description 'tcp:::accept-refused ' matched 1 probes
^C

  192.168.1.109                                         23              2
```

This shows that the 192.168.1.109 host attempted two connections to local port 23 (telnet), which were rejected. This one-liner could be used to detect port scans, which would appear as a host attempting to connect to numerous different ports.

What Am I Connecting To?

The `connect-established` probe traces active TCP connection established events, which are typically from local software establishing a connection to a remote server. Here a one-liner summarizes these events with remote host address and remote port:

```
solaris# dtrace -n 'tcp:::connect-established
{ @[args[3]->tcps_raddr , args[3]->tcps_rport] = count(); }'
dtrace: description 'tcp:::connect-established ' matched 1 probes
^C

  192.168.1.1                                           22              1
  192.168.1.3                                           80              2
```

During tracing, there were two outbound connections to 192.168.1.3 port 80 (HTTP.)

TCP Received Packets by Remote Address Summary

This one one-liner counts packets received by remote host address. The `receive` probe is used, and the remote host address is identified by the source address in the IP header `args[2]->ip_saddr`. This could also be derived from the TCP state information as `args[3]->tcps_raddr`.

```
solaris# dtrace -n 'tcp:::receive { @addr[args[2]->ip_saddr] = count(); }'
dtrace: description 'tcp:::receive ' matched 5 probes
^C

  127.0.0.1                                                         7
  fe80::214:4fff:fe8d:59aa                                         14
  192.168.2.30                                                     43
  192.168.1.109                                                    44
  192.168.2.88                                                   3722
```

While tracing, there were 3,722 TCP packets received from host 192.168.2.88. There were also 14 TCP packets received from an IPv6 host, fe80::214:4fff:fe8d:59aa.

TCP Received Packets by Local Port Summary

Similar to the previous one-liner, but this time the local port is traced by examining the destination port in the TCP header, `args[4]->tcp_dport`. This is also available in TCP state information as `args[3]->tcps_lport`.

```
solaris# dtrace -n 'tcp:::receive { @[args[4]->tcp_dport] = count(); }'
dtrace: description 'tcp:::receive ' matched 5 probes
^C

      42303                3
      42634                3
       2049               27
      40648               36
         22              162
```

While tracing, most of the received packets were for port 22 (SSH). The higher-numbered ports may be used by outbound (TCP active) connections.

Sent IP Payload Size Distributions

This one-liner prints distribution plots of IP payload size by remote host for TCP sends:

```
solaris# dtrace -n 'tcp:::send
{ @[args[2]->ip_daddr] = quantize(args[2]->ip_plength); }'
dtrace: description 'tcp:::send ' matched 3 probes
^C

  192.168.1.109
           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    14
             128 |@@@                                      1
             256 |                                         0

  192.168.2.30
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@                     7
              64 |@@@@@@@@@                                3
             128 |@@@                                      1
             256 |@@@@@@                                   2
             512 |@@@                                      1
            1024 |                                         0
```

The distribution shows the IP payload size, which includes the TCP header and the TCP data payload. To trace the actual TCP payload size, see the following one-liner.

Sent TCP Bytes Summary

This one-liner summarizes TCP sent payload bytes. This is determined by subtracting the TCP header offset from the IP payload length:

```
solaris# dtrace -n 'tcp:::send { @bytes = sum(args[2]->ip_plength -
args[4]->tcp_offset); }'
dtrace: description 'tcp:::send ' matched 3 probes
^C
      1004482
```

While tracing, 1,004,482 bytes of TCP payload was sent. This one-liner can be combined with others to provide this data by host, by port, every second, and so on.

TCP Events by Type Summary

This one-liner simply traces TCP probes by probe name:

```
solaris# dtrace -n 'tcp::: { @[probename] = count(); }'
dtrace: description 'tcp::: ' matched 41 probes
^C

  accept-established                                                2
  state-change                                                     12
  send                                                            103
  receive                                                         105
```

This can be used to compare the number of established connections with the number of sent and received TCP packets.

udp Provider Examples

The udp provider is demonstrated in the following examples.

UDP Sent Packets by Remote Port Summary

This one-liner counts UDP sent packets by the destination port:

```
solaris# dtrace -n 'udp:::send { @[args[4]->udp_dport] = count(); }'
dtrace: description 'udp:::send ' matched 5 probes
^C

    53                3
```

During tracing, there were three sent UDP packets to remote port 53 (DNS). The destination address `args[2]->ip_daddr` can be added to the aggregation to include the remote host, which in this case will identify the remote DNS servers queried.

Scripts

Table 6-9 summarizes the scripts that follow and the providers they use.

Table 6-9 Network Script Summary

Script	Protocol	Description	Provider
`soconnect.d`	Socket	Traces client socket `connect()`s showing process and host	syscall
`soaccept.d`	Socket	Traces server socket `accept()`s showing process and host	syscall
`soclose.d`	Socket	Traces socket connection duration: `connect()` to `close()`	syscall
`socketio.d`	Socket	Shows socket I/O by process and type	syscall
`socketiosort.d`	Socket	Shows socket I/O by process and type, sorted by process	syscall

continues

Table 6-9 Network Script Summary (*Continued*)

Script	Protocol	Description	Provider
`so1stbyte.d`	Socket	Traces connection and first-byte latency at the socket layer	syscall
`sotop.d`	Socket	Status tool to show top busiest sockets	syscall
`soerror.d`	Socket	Identifies socket errors	syscall
`ipstat.d`	IP	IP statistics every second	mib
`ipio.d`	IP	IP send/receive snoop	ip
`ipproto.d`	IP	IP encapsulated prototype summary	ip
`ipfbtsnoop.d`	IP	Trace IP packets: demonstration of fbt tracing	fbt
`tcpstat.d`	TCP	TCP statistics every second	mib
`tcpaccept.d`	TCP	Summarizes inbound TCP connections	tcp
`tcpacceptx.d`	TCP	Summarizes inbound TCP connections, resolve host names	tcp
`tcpconnect.d`	TCP	Summarizes outbound TCP connections	tcp
`tcpioshort.d`	TCP	Traces TCP send/receives live with basic details	tcp
`tcpio.d`	TCP	Traces TCP send/receives live with flag translation	tcp
`tcpbytes.d`	TCP	Sums TCP payload bytes by client and local port	tcp
`tcpsize.d`	TCP	Shows TCP send/receive I/O size distribution	tcp
`tcpnmap.d`	TCP	Detects possible TCP port scan activity	tcp
`tcpconnlat.d`	TCP	Measures TCP connection latency by remote host	tcp
`tcp1stbyte.d`	TCP	Measures TCP first byte latency by remote host	tcp
`tcp_rwndclosed.d`	TCP	Identifies TCP receive window zero events, with latency	tcp
`tcpfbtwatch.d`	TCP	Watches inbound TCP connections	fbt
`tcpsnoop.d`	TCP	Traces TCP I/O with process details	fbt
`udpstat.d`	UDP	UDP statistics every second	mib

Table 6-9 Network Script Summary (*Continued*)

Script	Protocol	Description	Provider
`udpio.d`	UDP	Traces UDP send/receives live with basic details	udp
`icmpstat.d`	ICMP	ICMP statistics every second	mib
`icmpsnoop.d`	ICMP	Traces ICMP packets with details	fbt
`superping.d`	ICMP	Improves accuracy of ping's round trip times	mib
`xdrshow.d`	XDR	Shows XDR calls and calling functions	fbt
`macops.d`	Ethernet	Counts MAC layer operations by interface and type	fbt
`ngesnoop.d`	Ethernet	Traces nge Ethernet events live	fbt
`ngelink.d`	Ethernet	Traces changes to nge link status	fbt

The fbt provider is considered an "unstable" interface, because it instruments a specific operating system version. For this reason, scripts that use the fbt provider may require changes to match the version of the software you are using. These scripts have been included here as examples of D programming and of the kind of data that DTrace can provide for each of these topics. See Chapter 12, Kernel, for more discussion about using the fbt provider.

Socket Scripts

Sockets are a standard interface of communication endpoints for application programming. Since sockets are created, read, and written using the system call interface, the `syscall` provider can be used to trace socket activity, which allows the application process ID responsible for socket activity to be identified, because it is still on-CPU during the system calls. The user stack backtrace can also be examined to show why an application is performing socket I/O. Figure 6-3 shows the typical application socket I/O flow.

The socket layer can be traced using the stable `syscall` provider, as shown in the one-liners. In the future, there may also be a stable socket provider available. The internals of the kernel socket implementation may be studied using the fbt provider, which can provide the most detailed view, at the cost of stability of the D scripts.

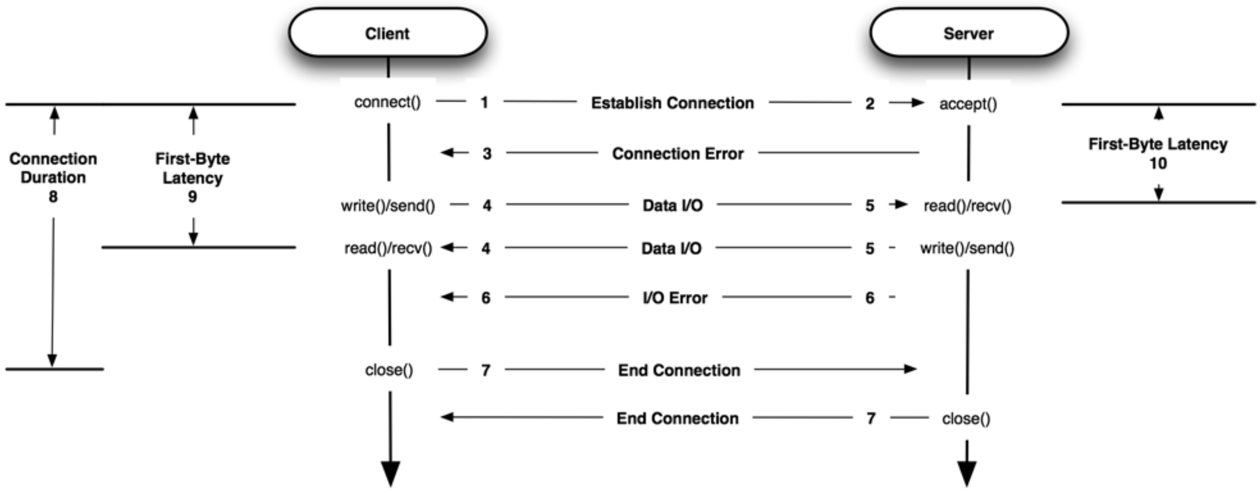


Figure 6-3 Socket flow diagram

Table 6-10 Use DTrace to Answer

	Question	Scripts
1	What outbound connections are occurring? To which server and port? `connect()` time? Why are applications performing connections, stack trace?	`soconnect.d`, one-liners
2	What inbound connections are being established?	`soaccept.d`
3	Connection errors	`soerror.d`
4	What client socket I/O is occurring, read and write bytes? By which process? What is performing the most socket I/O?	`socketio.d`, `sotop.d`
5	What server socket I/O is occurring, read and write bytes, by which processes? Client? Which process is performing the most socket I/O?	`socketiod`, `sotop.d`
6	Socket I/O errors	`soerror.d`
7	Who ends connections, and why? User-level stack trace?	Exercises
8	What is the duration of connections, with server and port details?	`soclose.d`
9	What is the time from connect to the first payload byte from the server?	`so1stbyte.d`
10	What is the time from accept to the first payload byte from the client?	`serv1stbyte.d`

soconnect.d

Applications execute the `connect()` system call on sockets to connect with remote peers. Tracing `connect()` calls on clients will show what network sessions are being established, as well as details such as latency. It is intended to be run on the client systems performing the connections.

Script

The `connect.d` script traces the `connect()` socket call from the `syscall` provider and extracts connection details from the arguments to connect. Since `arg1` may be a pointer to a struct `sockaddr_in` for AF_INET or `sockaddr_in6` for AF_INET6, which reside in user address space, to read the members, the entire structure must first be copied to kernel memory (`copyin()`). To know which structure type it is, we start by assuming it is struct `sockaddr_in`, copy it in, and then examine the address family. If this shows that it was AF_INET6, we recopy the data in as `sockaddr_in6`. This trick works because the address family member is at the start of both structs and is the same data type: a short.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

/* If AF_INET and AF_INET6 are "Unknown" to DTrace, replace with numbers: */
inline int af_inet = AF_INET;
inline int af_inet6 = AF_INET6;

dtrace:::BEGIN
{
        /* Add translations as desired from /usr/include/sys/errno.h */
        err[0]             = "Success";
        err[EINTR]         = "Interrupted syscall";
        err[EIO]           = "I/O error";
        err[EACCES]        = "Permission denied";
        err[ENETDOWN]      = "Network is down";
        err[ENETUNREACH]   = "Network unreachable";
        err[ECONNRESET]    = "Connection reset";
        err[ECONNREFUSED]  = "Connection refused";
        err[ETIMEDOUT]     = "Timed out";
        err[EHOSTDOWN]     = "Host down";
        err[EHOSTUNREACH]  = "No route to host";
        err[EINPROGRESS]   = "In progress";

        printf("%-6s %-16s %-3s %-16s %-5s %8s %s\n", "PID", "PROCESS", "FAM",
            "ADDRESS", "PORT", "LAT(us)", "RESULT");
}

syscall::connect*:entry
{
        /* assume this is sockaddr_in until we can examine family */
        this->s = (struct sockaddr_in *)copyin(arg1, sizeof (struct sockaddr));
        this->f = this->s->sin_family;
}

syscall::connect*:entry
/this->f == af_inet/
{
        self->family = this->f;
        self->port = ntohs(this->s->sin_port);
        self->address = inet_ntop(self->family, (void *)&this->s->sin_addr);
        self->start = timestamp;
}

syscall::connect*:entry
/this->f == af_inet6/
{
        /* refetch for sockaddr_in6 */
        this->s6 = (struct sockaddr_in6 *)copyin(arg1,
            sizeof (struct sockaddr_in6));
        self->family = this->f;
        self->port = ntohs(this->s6->sin6_port);
        self->address = inet_ntoa6((in6_addr_t *)&this->s6->sin6_addr);
        self->start = timestamp;
}

syscall::connect*:return
/self->start/
{
        this->delta = (timestamp - self->start) / 1000;
        this->errstr = err[errno] != NULL ? err[errno] : lltostr(errno);
        printf("%-6d %-16s %-3d %-16s %-5d %8d %s\n", pid, execname,
            self->family, self->address, self->port, this->delta, this->errstr);
        self->family = 0;
```

```
66          self->address = 0;
67          self->port = 0;
68          self->start = 0;
69  }
Script soconnect.d
```

Table 6-11 Example AF_INET and AF_INET6 Values

Operating System	AF_INET	AF_INET6	Source
Solaris 10	2	26	`/usr/include/sys/socket.h`
OpenSolaris	2	26	`/usr/include/sys/socket.h`
Mac OS X 10.6	2	30	`bsd/sys/socket.h`
FreeBSD 8.0	2	28	`sys/socket.h`

Some operating system versions will have AF_INET and AF_INET6 defined for use by DTrace (they were added for the network providers), which are needed for this script. If they are not known (for example, on current Solaris 10, Mac OS X, and FreeBSD), the script will produce the error “failed to resolve AF_INET: Unknown variable name.” If that happens, edit lines 7 and 8 to replace AF_INET and AF_INET6 to be the correct values for your operating system (or, use the C-preprocessor to source them). These values may change; Table 6-11 shows recent values as a hint, but these should be double-checked before use.

The use of `this->f` instead of just allocating `self->family` to begin with is to avoid allocating a thread-local variable that would later need cleaning up if it didn’t match the predicates on lines 38 and 47.

Connection latency is calculated as the time from `syscall::connect*:entry` to `syscall::connect*:return`. Calculating delta times for socket operations at the system call layer is easy, since the system call occurs in process/thread context and thread-local variables (`self->`) can be used. The `connect:return` function also allows the error status to be checked. A partial table of error codes to strings is in the `dtrace:::BEGIN` block for translation.

On older versions of Solaris that do not have `inet_ntop()` available in DTrace, and for Mac OS X that also currently lacks `ntohs()`, the `syscall::connect*:entry` action can be rewritten like this:

```
37  syscall::connect*:entry
38  /this->f == af_inet/
39  {
40          self->family = this->f;
41
42          /* Convert port to host byte order without ntohs() being available. */
```

continues

```
        self->port = (this->s->sin_port & 0xFF00) >> 8;
        self->port |= (this->s->sin_port & 0xFF) << 8;

        /*
         * Convert an IPv4 address into a dotted quad decimal string.
         * Until the inet_ntoa() functions are available from DTrace, this is
         * converted using the existing strjoin() and lltostr().  It's done in
         * two parts to avoid exhausting DTrace registers in one line of code.
         */
        this->a = (uint8_t *)&this->s->sin_addr;
        this->addr1 = strjoin(lltostr(this->a[0] + 0ULL), strjoin(".",
            strjoin(lltostr(this->a[1] + 0ULL), ".")));
        this->addr2 = strjoin(lltostr(this->a[2] + 0ULL), strjoin(".",
            lltostr(this->a[3] + 0ULL)));
        self->address = strjoin(this->addr1, this->addr2);

        self->start = timestamp;
}
```

Script soconnect_mac.d

This replacement `inet_ntop()` code is for IPv4 address (address family is `AF_INET`). It produces the IPv4 address string manually, without assuming that the `inet_ntop()` function is available (it may not be, depending on your DTrace version). Similar (and longer) code could be written for IPv6 to produce an eight 16-bit number representation of the form x:x:x:x:x:x:x:x; compact form (see RFC1924) is expected to be difficult to produce in this way (which is why the `inet*` functions are needed).

The replacement `ntohs()` code is for little-endian systems such as Mac OS X on Intel. For big-endian systems, conversion isn't necessary.

See the `ipfbtsnoop.d` script in the "IP Scripts" section for another example of IPv4 manual stringification, with a reusable macro.

Examples

The following examples demonstrate the use of the `soconnect.d` script.

Application Connect Snooping. The following example shows `soconnect.d` executed on a Solaris client. The first four lines show two successful SSH connections and then two unsuccessful Telnet connections; the second was interrupted (Ctrl-C) after waiting 2.8 seconds for it to connect.

Then the Firefox Web browser loaded the *www.sun.com* Web site, and we can see the DNS queries from the nscd process (Name Service Cache Daemon) to port 53, followed by HTTP requests from `firefox-bin` to port 80.

```
client# soconnect.d
PID    PROCESS           FAM ADDRESS          PORT   LAT(us) RESULT
54677  ssh               2   192.168.2.156    22         210 Success
54730  ssh               2   192.168.1.3      22         436 Success
```

```
  54878  telnet           2   192.168.2.156    23          321 Connection refused
  54931  telnet           2   192.168.1.3      23      2835157 Interrupted syscall
  356    nscd             2   192.168.1.5      53           54 Success
  356    nscd             2   192.168.1.5      53           52 Success
  356    nscd             2   192.168.1.5      53           38 Success
  356    nscd             2   192.168.1.5      53           37 Success
  22642  firefox-bin      2   72.5.124.61      80          138 In progress
  22642  firefox-bin      2   72.5.124.61      80           64 In progress
  356    nscd             2   192.168.1.5      53           55 Success
  356    nscd             2   192.168.1.5      53           53 Success
  22642  firefox-bin      2   80.67.66.55      80          109 In progress
  22642  firefox-bin      2   80.67.66.55      80           45 In progress
  356    nscd             2   192.168.1.5      53           55 Success
  356    nscd             2   192.168.1.5      53           43 Success
  22642  firefox-bin      2   66.235.132.118   80          110 In progress
  10613  nfsmapid         2   192.168.1.5      53           56 Success
  356    nscd             2   192.168.1.5      53           55 Success
  356    nscd             2   192.168.1.5      53           53 Success
  5002   elinks           2   74.86.31.159     80          116 In progress
  55555  ssh              2   10.1.0.23        22        38003 Success
  55179  ssh              2   10.1.0.25        22    224659402 Timed out
  10613  nfsmapid         2   192.168.1.5      53           54 Success
  ^C
```

The penultimate line was an SSH to an offline host, which took 225 seconds before the connection timed out.

Port Scanning. Here the `nmap` port scanner was used to perform a TCP Connect scan on a local server:

The connection attempts from `nmap` can be observed clearly in the output.

```
  client# soconnect.d
  PID    PROCESS          FAM ADDRESS          PORT   LAT(us) RESULT
  911287 nmap             2   192.168.1.5      53          79 Success
  911287 nmap             2   192.168.2.145    443         67 In progress
  911287 nmap             2   192.168.2.145    22          51 In progress
  911287 nmap             2   192.168.2.145    3389        19 In progress
  911287 nmap             2   192.168.2.145    389         48 In progress
  911287 nmap             2   192.168.2.145    80          19 In progress
  911287 nmap             2   192.168.2.145    1723        37 In progress
  911287 nmap             2   192.168.2.145    23          19 In progress
  911287 nmap             2   192.168.2.145    21          19 In progress
  911287 nmap             2   192.168.2.145    113         41 In progress
  911287 nmap             2   192.168.2.145    53          20 In progress
  911287 nmap             2   192.168.2.145    636         26 In progress
  911287 nmap             2   192.168.2.145    554         19 In progress
  911287 nmap             2   192.168.2.145    25          36 In progress
  911287 nmap             2   192.168.2.145    256         36 In progress
  911287 nmap             2   192.168.2.145    14922       36 In progress
  911287 nmap             2   192.168.2.145    27471       35 In progress
  911287 nmap             2   192.168.2.145    11814       36 In progress
  911287 nmap             2   192.168.2.145    25072       35 In progress
  911287 nmap             2   192.168.2.145    48457       19 In progress
  [...]
```

soaccept.d

Inbound socket connections can be traced on the server by probing `accept()`.

Script

Just as in soconnect.d, both IPv4 and IPv6 connections are processed by first assuming IPv4, copying in the sockaddr, and checking the address family. If it was IPv6 after all, the sockaddr is copied in again as sockaddr_in6.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

/* If AF_INET and AF_INET6 are "Unknown" to DTrace, replace with numbers: */
inline int af_inet = AF_INET;
inline int af_inet6 = AF_INET6;

dtrace:::BEGIN
{
        /* Add translations as desired from /usr/include/sys/errno.h */
        err[0]            = "Success";
        err[EINTR]        = "Interrupted syscall";
        err[EIO]          = "I/O error";
        err[EAGAIN]       = "Resource temp unavail";
        err[EACCES]       = "Permission denied";
        err[ECONNABORTED] = "Connection aborted";
        err[ECONNRESET]   = "Connection reset";
        err[ETIMEDOUT]    = "Timed out";
        err[EINPROGRESS]  = "In progress";

        printf("%-6s %-16s %-3s %-16s %-5s %8s %s\n", "PID", "PROCESS", "FAM",
            "ADDRESS", "PORT", "LAT(us)", "RESULT");
}

syscall::accept*:entry
{
        self->sa = arg1;
        self->start = timestamp;
}

syscall::accept*:return
/self->sa/
{
        this->delta = (timestamp - self->start) / 1000;
        /* assume this is sockaddr_in until we can examine family */
        this->s = (struct sockaddr_in *)copyin(self->sa,
            sizeof (struct sockaddr_in));
        this->f = this->s->sin_family;
}

syscall::accept*:return
/this->f == af_inet/
{
        this->port = ntohs(this->s->sin_port);
        this->address = inet_ntoa((ipaddr_t *)&this->s->sin_addr);
        this->errstr = err[errno] != NULL ? err[errno] : lltostr(errno);
        printf("%-6d %-16s %-3d %-16s %-5d %8d %s\n", pid, execname,
            this->f, this->address, this->port, this->delta, this->errstr);
}

syscall::accept*:return
/this->f == af_inet6/
{
        /* refetch for sockaddr_in6 */
```

```
        this->s6 = (struct sockaddr_in6 *)copyin(self->sa,
            sizeof (struct sockaddr_in6));
        this->port = ntohs(this->s6->sin6_port);
        this->address = inet_ntoa6((in6_addr_t *)&this->s6->sin6_addr);
        this->errstr = err[errno] != NULL ? err[errno] : lltostr(errno);
        printf("%-6d %-16s %-3d %-16s %-5d %8d %s\n", pid, execname,
            this->f, this->address, this->port, this->delta, this->errstr);
}

syscall::accept*:return
/self->start/
{
        self->sa = 0; self->start = 0;
}
```

Script soaccept.d

To print IPv4 and IPv6 addresses as strings, the `inet_ntoa()` and `inet_ntoa6()` DTrace functions were used. If currently unavailable in your version of DTrace, process it manually (see `soconnect.d`). The `PORT` number printed is the remote port, not the local port.

Example

Here an inbound `ssh` connection was found, which used the remote port 44364. The `netstat` command was used to see what local port that connected to: port 22.

```
server# soaccept.d
PID    PROCESS          FAM ADDRESS          PORT   LAT(us) RESULT
8491   httpd            26  192.168.1.109    45416       41 Success
1111   sshd             26  192.168.1.109    63485       31 Success
8494   httpd            26  192.168.1.109    38862       19 Success
8490   httpd            26  192.168.1.109    55298       13 Success
1161   httpd            2   192.168.1.109    0           49 Success
1158   httpd            2   192.168.1.109    0           37 Success
1111   sshd             26  192.168.1.109    44364       40 Success
^C
server# netstat -an | grep 44364
192.168.2.145.22     192.168.1.109.44364  49640      0 1049740      0
ESTABLISHED
```

soclose.d

This script measures the duration of socket connections of the Internet Protocol type and prints details including the target address and port. It is intended to be run on the client host performing the connections.

Script

The duration of the connection is measured from the `connect()` to the `close()` of the socket file descriptor:

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   /* If AF_INET and AF_INET6 are "Unknown" to DTrace, replace with numbers: */
7   inline int af_inet = AF_INET;
8   inline int af_inet6 = AF_INET6;
9
10  dtrace:::BEGIN
11  {
12          printf("  %-6s %-16s %-3s %-16s %-5s %s\n", "PID", "PROCESS", "FAM",
13              "ADDRESS", "PORT", "DURATION(sec)");
14  }
15
16  syscall::connect*:entry
17  {
18          this->s = (struct sockaddr_in *)copyin(arg1, sizeof (struct sockaddr));
19          this->f = this->s->sin_family;
20  }
21
22  syscall::connect*:entry
23  /this->f == af_inet || this->f == af_inet6/
24  {
25          self->family[arg0] = this->f;
26          self->port[arg0] = ntohs(this->s->sin_port);
27          self->address[arg0] = inet_ntop(this->s->sin_family,
28              (void *)&this->s->sin_addr);
29          self->start[arg0] = timestamp;
30  }
31
32  syscall::close:entry
33  /self->start[arg0]/
34  {
35          this->delta = (timestamp - self->start[arg0]) / 1000;
36          this->sec = this->delta / 1000000;
37          this->ms = (this->delta - (this->sec * 1000000)) / 1000;
38          printf("  %-6d %-16s %-3d %-16s %-5d %d.%03d\n", pid, execname,
39              self->family[arg0], self->address[arg0], self->port[arg0],
40              this->sec, this->ms);
41          self->family[arg0] = 0;
42          self->address[arg0] = 0;
43          self->port[arg0] = 0;
44          self->start[arg0] = 0;
45  }
```

Script soclose.d

Time stamps and other variables are keyed on the file descriptor, in case the process opens multiple connections in parallel.

For readability, the duration is printed as "<seconds>.<milliseconds>" with up to three decimal places. If floating-point operators existed in DTrace, this would just require printing seconds as a float using a `%.3f` operand for `printf()`. `printf()` supports the format operand, but the operator to calculate the float is not supported. As a workaround, lines 35 to 37 calculate both the seconds and milliseconds components as `this->sec` and `this->ms`; these are then printed—the millisecond component with up to three leading zeros (`%03d`)—on line 38.

This script can be modified similarly to `soconnect.d` so that it executes on older Solaris versions or Mac OS X.

Example

The `soclose.d` script was executed on a Solaris desktop while several `ssh` commands were run, and a Web site loaded in the Firefox Web browser:

```
client# soclose.d
  PID    PROCESS          FAM ADDRESS          PORT  DURATION(sec)
  739286 ssh              2   192.168.1.188    22    3.625
  708951 nscd             2   192.168.1.5      53    0.000
  708951 nscd             2   192.168.1.5      53    0.316
  708951 nscd             2   192.168.1.5      53    0.824
  708951 nscd             2   192.168.1.5      53    0.382
  608440 firefox-bin      2   66.235.132.118   80    15.964
  708951 nscd             2   192.168.1.5      53    0.000
  739475 ssh              2   192.168.1.3      22    10.957
  608440 firefox-bin      2   72.5.124.61      80    63.464
```

The duration of the `ssh` sessions of 3.625 and 10.957 seconds are indeed the time that those `ssh` sessions were logged in. The long-duration connections from `firefox-bin` are evidence of HTTP keep-alives.

socketio.d

Summarizing the socket I/O calls that are occurring is a starting point for investigating socket behavior and performance, and it may directly identify load-related problems.

Script

`socketio.d` is a high-level script that may also be useful for further customizations:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing Socket I/O... Hit Ctrl-C to end.\n");
}

syscall::read*:entry,
syscall::write*:entry,
syscall::send*:entry,
syscall::recv*:entry
/fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>"/
{
         @[execname, pid, probefunc] = count();
}
```

continues

```
18
19  dtrace:::END
20  {
21          printf("  %-16s %-8s %-16s %10s\n", "PROCESS", "PID", "SYSCALL",
22              "COUNT");
23          printa("  %-16s %-8d %-16s %@10d\n", @);
24  }
```

Script socketio.d

Line 14 has been written so that this script executes on both Solaris and Mac OS X, by testing either method for identifying sockets in an OR (||) statement (the statement is `Solaris socket OR Mac OS X socket`).

Example

This script was executed on a Solaris workstation with a Java application performing 10,000 TCP sends:

```
solaris# socketio.d
Tracing Socket I/O... Hit Ctrl-C to end.
^C
  PROCESS          PID      SYSCALL               COUNT
  ssh              864116   write                     1
  sshd             942634   read                      1
  FvwmPager        701861   write                     2
  ssh              864116   read                      4
  xclock           785004   write                     4
  FvwmIconMan      701860   write                     5
  FvwmPager        701865   write                     5
  FvwmPager        701865   read                      7
  sshd             942634   write                     7
  firefox-bin      272642   write                     8
  soffice.bin      453667   read                      8
  soffice.bin      453667   write                     8
  fvwm2            701854   write                    25
  gnome-terminal   701876   write                    37
  firefox-bin      272642   read                     40
  fvwm2            701854   read                     41
  gnome-terminal   701876   read                     49
  Xorg             614773   writev                  100
  Xorg             614773   read                    207
  java             440474   send                  10000
```

The `java` application and socket I/O call was identified with the correct count. All socket I/O was traced (this is not filtering on protocol family types AF_INET/AF_INET6), including socket I/O from various daemons that drive the desktop environment (FVWM2).

socketiosort.d

The previous output of `socketio.d` sorted the output by count. At times you may find it useful to group applications together, but doing this in the output can be a nontrivial task—something suited to postprocessing using, for example, Perl. DTrace provides for changing the default sort key based on your needs.

Script

The first 13 lines are the same as socketio.d; and then the script changes on line 14:

```
14  /fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>"/
15  {
16          @num[execname, probefunc, pid] = count();
17          @pid[execname, probefunc, pid] = max(pid);
18          @pid["--------------", "------", pid] = max(pid);
19  }
20
21  dtrace:::END
22  {
23          printf("  %-8s %-16s %-16s %10s\n", "PID", "PROCESS", "SYSCALL",
24              "COUNT");
25          setopt("aggsortpos", "0");
26          printa("  %@-8d %-16s %-16s %@10d\n", @pid, @num);
27  }
```

Script socketiosort.d

DTrace has the aggsortpos option, which controls selecting which output column to sort by, provided it is an aggregation value. To group the processes together, we need to sort by either the PID or the process name, which were aggregation keys, not values. As a workaround, the PID column is changed into the @pid aggregation, which allows sorting by PID. The extra pid key is discarded and prevents the max() function from ignoring some PIDs.

Example

The output is easier to read by process:

```
solaris# socketiosort.d
Tracing Socket I/O... Hit Ctrl-C to end.
^C
  PID      PROCESS          SYSCALL               COUNT
  272642   --------------   ------                    0
  272642   firefox-bin      write                    28
  272642   firefox-bin      read                    142
  439751   --------------   ------                    0
  439751   java             send                  10000
  453667   --------------   ------                    0
  453667   soffice.bin      read                     24
  453667   soffice.bin      write                    24
  614773   --------------   ------                    0
  614773   Xorg             writev                  109
  614773   Xorg             read                    368
  701854   --------------   ------                    0
  701854   fvwm2            write                    25
```

continues

```
   701854   fvwm2            read                     40
   701860   --------------   ------                    0
   701860   FvwmIconMan      write                     5
   701861   --------------   ------                    0
   701861   FvwmPager        write                     2
   701865   --------------   ------                    0
   701865   FvwmPager        write                     6
   701865   FvwmPager        read                     10
   701876   --------------   ------                    0
   701876   gnome-terminal   read                     60
   701876   gnome-terminal   write                    62
   785004   --------------   ------                    0
   785004   xclock           write                    12
   864116   --------------   ------                    0
   864116   ssh              write                     1
   864116   ssh              read                      5
   942634   --------------   ------                    0
   942634   sshd             read                      1
   942634   sshd             write                     8
```

so1stbyte.d

Connection latency and first-byte latency can identify different characteristics of network connections. Connection latency was observed earlier with the `soconnect.d` script; first-byte latency is the time from when a connection is established to when the first data byte is read. This time includes service initialization and packet round-trip time.

Script

This script matches the first-byte event on line 37, which checks that the return value of the `read()` or `recv()` syscall is greater than zero (`arg0` > 0).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8          printf("  %6s %-16s %6s  %14s %14s  %8s\n", "PID", "PROCESS", "PORT",
9              "CONNECT(us)", "1stBYTE(us)", "BYTES");
10  }
11
12  syscall::connect*:entry
13  {
14          this->s = (struct sockaddr_in *)copyin(arg1, sizeof (struct sockaddr));
15          self->port = (this->s->sin_port & 0xFF00) >> 8;
16          self->port |= (this->s->sin_port & 0xFF) << 8;
17          self->start = timestamp;
18          self->connected = 0;
19  }
20
21  syscall::connect*:return
22  {
23          self->connection = (timestamp - self->start) / 1000;
24          self->start = 0;
```

```
        self->connected = timestamp;
}

syscall::read*:entry, syscall::recv*:entry
/(fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>") &&
    self->connected/
{
        self->socket = 1;
}

syscall::read*:return, syscall::recv*:return
/self->socket && arg0 > 0/
{
        this->firstbyte = (timestamp - self->connected) / 1000;
        printf("  %6d %-16s %6d  %14d %14d  %8d\n", pid, execname, self->port,
            self->connection, this->firstbyte, arg0);
        self->connected = 0;
        self->socket = 0;
        self->port = 0;
}
```

Script so1stbyte.d

Examples

`so1stbyte.d` examples are presented in this section.

Loading a Web Site. On a Solaris workstation, the Web site *www.solarisinternals .com* was loaded in the Firefox Web browser. The `so1stbyte.d` script showed various first-byte latencies as components of the Web site loaded and DNS requests handled by `nscd` (Name Service Cache Daemon):

```
client# so1stbyte.d
     PID PROCESS             PORT    CONNECT(us)    1stBYTE(us)     BYTES
  708951 nscd                  53             54          44116        86
  708951 nscd                  53             54            578       102
  708951 nscd                  53            314            400       102
  708951 nscd                  53             38          28668       110
  608440 firefox-bin           80            114         148059       637
  708951 nscd                  53             53          35862       136
  708951 nscd                  53             51          36211       254
  608440 firefox-bin           80             98          40222       349
  608440 firefox-bin           80             62          15248      2920
  708951 nscd                  53             54          59906        79
  708951 nscd                  53             52            732        92
  708951 nscd                  53             35            475        92
  708951 nscd                  53             27         102845       196
  608440 firefox-bin           80             26         261675      3282
  708951 nscd                  53             53          44037       214
  708951 nscd                  53             52            439       214
  608440 firefox-bin           80            102          71871      2705
^C
```

Experiments. The following experiments were performed to compare changes in connect and first-byte latency.

Here we compare `ssh(1)` vs. `telnet(1)`.

```
client# so1stbyte.d
     PID PROCESS              PORT      CONNECT(us)     1stBYTE(us)      BYTES
  712248 ssh                    22              327           21259          1
  712265 ssh                    22              291           18631          1
  712284 ssh                    22              644           23384          1
  713200 telnet                 23             2103          135989          3
  713249 telnet                 23              339           89227          3
  713278 telnet                 23              345           97422          3
```

The first three connections used `ssh`; the next three used `telnet` (see the `PROCESS` column). Note the increase in first-byte latency for `telnet`. The extra latency is likely because of `telnet` being serviced by `inetd` (inet daemon) spawning a new process, `in.telnetd`, whereas the `sshd` (`ssh` daemon) is always running (this chain of events can be DTraced directly on the remote host to confirm). Encryption is unlikely to play a role in first-byte latency, because the first byte from `ssh` is the unencrypted SSH version string.

Here we compare Wi-Fi vs. Ethernet:

```
client# so1stbyte.d
     PID PROCESS              PORT      CONNECT(us)     1stBYTE(us)      BYTES
  716019 ssh                    22           154559           20099          1
  716034 ssh                    22           385660           17957          1
  716053 ssh                    22           321607           17915          1
  717879 ssh                    22              527           19878          1
  717896 ssh                    22              633           18343          1
  717913 ssh                    22              658           68770          1
```

The first three connections were to a host over Wi-Fi; the second three were to the same host but over Ethernet. The Wi-Fi connections have a dramatically higher connection latency.

Here we compare local vs. distant:

```
client# so1stbyte.d
     PID PROCESS              PORT      CONNECT(us)     1stBYTE(us)      BYTES
  721282 ssh                    22              408           16345          1
  721299 ssh                    22              406           32970          1
  721314 ssh                    22              300           26488          1
  721385 ssh                    22           172992          175863          1
  721402 ssh                    22           174349          176329          1
  721419 ssh                    22           173050          176192          1
```

The first three `ssh` connections were to a local host in San Francisco. The last three were to a host in Australia. Notice both connect and first-byte latencies exceed 170 ms. This is the round-trip time to the remote host (measured using the `ping` command).

sotop.d

Socket top[12] shows socket IOPS and throughput by process, along with CPU usage, refreshing the screen every second.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option destructive

syscall::read*:entry, syscall::recv*:entry
/fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>"/
{
       self->read = 1;
}

syscall::read*:return, syscall::recv*:return
/self->read/
{
        this->size = (int)arg0 > 0 ? arg0 : 0;
        @rc[execname, pid] = count();
        @rb[execname, pid] = sum(this->size);
        self->read = 0;
}

syscall::write*:entry, syscall::send*:entry
/fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>"/
{
        /* this under-counts writev() size (assumes iov_len is 1) */
        this->size = arg2;
        @wc[execname, pid] = count();
        @wb[execname, pid] = sum(this->size);
}

profile:::profile-100hz
{
        /* will sum %CPUs on multi-core systems */
        @cpu[execname, pid] = count();
}

profile:::tick-1sec
{
        normalize(@rb, 1024); normalize(@wb, 1024);
        system("clear");
        printf("  %-16s %-8s %8s %8s %10s %10s %8s\n", "PROCESS", "PID",
            "READS", "WRITES", "READ_KB", "WRITE_KB", "CPU");
        setopt("aggsortpos", "4"); setopt("aggsortrev", "4");
        printa("  %-16s %-8d %@8d %@8d %@10d %@10d %@8d\n",
            @rc, @wc, @rb, @wb, @cpu);
        trunc(@rc); trunc(@rb); trunc(@wc); trunc(@wb); trunc(@cpu);
}
```

Script sotop.d

12. top(1) is a popular process usage tool that was written by William LeFebvre.

Note that on line 42 we reverse sort the output by the CPU column. CPU shows the number of times the application was on a CPU, sampled at 100 Hertz by the `profile-100hz` probe. On multi-CPU systems with the application running on multiple CPUs concurrently, that count may be greater than 100 during a single second. This count could be converted into a percent CPU column (`%CPU`) if desired, by using an additional `normalize()` function on line 38 to divide `@cpu` by the online CPU count.[13]

The screen is cleared by calling `system("clear")`, which requires using the destructive option, set on line 4.

Apart from utility, this script demonstrates a different style of formatting status output (`top(1)`-like), which can be reused for other D scripts.

Example

```
solaris# sotop.d
  PROCESS          PID          READS   WRITES    READ_KB   WRITE_KB       CPU
  sched            0                0        0          0          0        51
  ttcp             158138       10462        0       9615          0        14
  gnome-terminal   701876          61       52          2         18        10
  Xorg             614773         422      197         51          0         8
  firefox-bin      608440           0        0          0          0         8
  operapluginwrapp 835656         132       95          3          1         4
  java             958443           0        0          0          0         1
  fsflush          3                0        0          0          0         1
  elinks           955002           0        0          0          0         1
  fvwm2            701854         107       68          2         17         0
  FvwmPager        701865          23       15          0          1         0
  soffice.bin      835606           2        2          0          0         0
  opera            835641           2        0          0          0         0
  FvwmIconMan      701860           0       16          0          2         0
  FvwmPager        701861           0        6          0          0         0
  xclock           785004           0        1          0          8         0
```

While `sotop.d` was running, the `ttcp` tool was used to receive network traffic. In the previous sample, `ttcp` was reading at 9.6MB/sec.

The top process, named `sched`, is the kernel (`kernel_task` on Mac OS X) and is likely to be the idle thread. The previous output shows a system that would be close to 51 percent idle.

13. The number of currently online CPUs should be provided as a stable built-in integer variable to DTrace for use for times like this. Until that exists, there are a few other ways to include this in a D script, including hard-coding it; passing it at the command line and using the `$1` macro variable; fetching it from a kernel variable (given that is an unstable interface), such as `` `ncpus_online `` on Solaris.

soerror.d

Errors are often interesting to monitor because they can reveal misconfigurations or software bugs. This script traces errors reported by socket-based system calls, by examining the return value for the system call. The output of this script simply means that a system call returned an error. The application may have processed this error correctly, and in some cases the error may have been expected and is normal.

Script

The error codes and short descriptions have been sourced from `/usr/include/sys/errno.h`. Only some of the errors are included in the following translation table; more can be added if desired:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        /* Add translations as desired from /usr/include/sys/errno.h */
        err[0]             = "Success";
        err[EACCES]        = "Permission denied";
        err[ECONNABORTED]  = "Connection abort";
        err[ECONNREFUSED]  = "Connection refused";
        err[ECONNRESET]    = "Connection reset";
        err[EHOSTDOWN]     = "Host down";
        err[EHOSTUNREACH]  = "No route to host";
        err[EINPROGRESS]   = "In progress";
        err[EINTR]         = "Interrupted syscall";
        err[EINVAL]        = "Invalid argument";
        err[EIO]           = "I/O error";
        err[ENETDOWN]      = "Network is down";
        err[ENETUNREACH]   = "Network unreachable";
        err[EPROTO]        = "Protocol error";
        err[ETIMEDOUT]     = "Timed out";
        err[EWOULDBLOCK]   = "Would block";

        printf("  %-6s %-16s %-10s %-4s %4s %4s  %s\n", "PID", "PROCESS",
            "SYSCALL", "FD", "RVAL", "ERR", "RESULT");
}

syscall::connect*:entry, syscall::accept*:entry,
syscall::getsockopt:entry, syscall::setsockopt:entry
{
        self->fd = arg0; self->ok = 1;
}

syscall::read*:entry, syscall::write*:entry,
syscall::send*:entry, syscall::recv*:entry
/fds[arg0].fi_fs == "sockfs" || fds[arg0].fi_name == "<socket>"/
{
        self->fd = arg0; self->ok = 1;
}

```

continues

```
43  syscall::so*:entry
44  {
45          self->ok = 1;
46  }
47
48  syscall::connect*:return, syscall::accept*:return,
49  syscall::read*:return, syscall::write*:return,
50  syscall::send*:return, syscall::recv*:return,
51  syscall::getsockopt:return, syscall::setsockopt:return
52  /errno != 0 && errno != EAGAIN && self->ok/
53  {
54          this->errstr = err[errno] != NULL ? err[errno] : lltostr(errno);
55          printf("  %-6d %-16s %-10s %-4d %4d %4d  %s\n", pid, execname, probefunc,
56              self->fd, arg0, errno, this->errstr);
57  }
58
59  syscall::so*:return
60  /errno != 0/
61  {
62          /* these syscalls (such as sockconfig) don't operate on socket fds */
63          this->errstr = err[errno] != NULL ? err[errno] : lltostr(errno);
64          printf("  %-6d %-16s %-10s %-4s %4d %4d  %s\n", pid, execname, probefunc,
65              "-", arg0, errno, this->errstr);
66  }
67
68  syscall::connect*:return, syscall::accept*:return,
69  syscall::read*:return, syscall::write*:return,
70  syscall::send*:return, syscall::recv*:return,
71  syscall::getsockopt:return, syscall::setsockopt:return,
72  syscall::so*:return
73  {
74          self->fd = 0; self->ok = 0;
75  }
```

Script soerror.d

All socket-related system calls, including read/write/send/recv to socket file descriptions, are traced and checked for errors.

Lines 30 to 46 checks various system calls that operate on file descriptors to see whether they are for sockets, setting a `self->ok` thread-local variable if they are. That is then checked on line 52, along with the built-in `errno` variable that contains the error code for the last system call. EAGAIN codes are skipped, because they can be a normal part of socket operation, not an error type we are interested in.

Lines 59 to 66 checks all socket system calls by matching their name as `so*` and checks that `errno` is set.

Example

Various socket errors are visible in the following output from `soerror.d`:

```
solaris# soerror.d
  PID    PROCESS          SYSCALL    FD    RVAL  ERR  RESULT
  810779 telnet           connect    4       -1  146  Connection refused
  808747 ssh              connect    4       -1    4  Interrupted syscall
  608440 firefox-bin      connect    10      -1  150  In progress
```

```
    608440 firefox-bin      connect    38      -1  150  In progress
    608440 firefox-bin      connect    10      -1  150  In progress
    608440 firefox-bin      connect    10      -1  150  In progress
    608440 firefox-bin      connect    10      -1  150  In progress
    608440 firefox-bin      connect    10      -1  150  In progress
    608440 firefox-bin      connect    10      -1  150  In progress
    808889 ttcp             read       0       -1    4  Interrupted syscall
    809183 ttcp             accept     3       -1    4  Interrupted syscall
    809206 ttcp             accept     3       -1    4  Interrupted syscall
  [...]
```

Reference

To understand these errors in more detail, consult the `/usr/include/sys/errno.h` file for the error number to code translations and the system call man page for the long descriptions: `connect(3SOCKET)`, `accept(3SOCKET)`, and so on. `errno.h` has the following format:

```
/usr/include/sys/errno.h:
[...]
#define ENETDOWN        127     /* Network is down */
#define ENETUNREACH     128     /* Network is unreachable */
#define ENETRESET       129     /* Network dropped connection because */
                                /* of reset */
#define ECONNABORTED    130     /* Software caused connection abort */
#define ECONNRESET      131     /* Connection reset by peer */
#define ENOBUFS         132     /* No buffer space available */
#define EISCONN         133     /* Socket is already connected */
#define ENOTCONN        134     /* Socket is not connected */
[...]
```

Table 6-12 lists some errors with their full descriptions.

Table 6-12 Socket System Call Error Descriptions

System Call	Error Code	Description
`connect()`	ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should `close(2)` the socket descriptor and issue another `socket(3SOCKET)` call to obtain a new descriptor before attempting another `connect()` call.
`connect()`	EINPROGRESS	The socket is nonblocking, and the connection cannot be completed immediately. You can use `select(3C)` to complete the connection by selecting the socket for writing.

continues

Table 6-12 Socket System Call Error Descriptions (*Continued*)

System Call	Error Code	Description
`connect()`	EINTR	The connection attempt was interrupted before any data arrived by the delivery of a signal. The connection, however, will be established asynchronously.
`connect()`	ENETUNREACH	The network is not reachable from this host.
`connect()`	EHOSTUNREACH	The remote host is not reachable from this host.
`connect()`	ETIMEDOUT	The connection establishment timed out without establishing a connection.
`accept()`	ECONNABORTED	The remote side aborted the connection before the `accept()` operation completed.
`accept()`	EINTR	The `accept()` attempt was interrupted by the delivery of a signal.
`accept()`	EPROTO	A protocol error has occurred; for example, the `STREAMS` protocol stack has not been initialized or the connection has already been released.
`accept()`	EWOULDBLOCK	The socket is marked as nonblocking, and no connections are present to be accepted.
`read()`	EINTR	A signal was caught during the read operation, and no data was transferred.
`write()`	EINTR	A signal was caught during the write operation, and no data was transferred.
`send()`	EINTR	The operation was interrupted by delivery of a signal before any data could be buffered to be sent.
`send()`	EMSGSIZE	The socket requires that the message be sent atomically and the message is too long.
`send()`	EWOULDBLOCK	The socket is marked nonblocking, and the requested operation would block. EWOULDBLOCK is also returned when sufficient memory is not immediately available to allocate a suitable buffer. In such a case, the operation can be retried later.
`recv()`	EINTR	The operation is interrupted by the delivery of a signal before any data is available to be received.
`recv()`	ENOSR	Insufficient `STREAMS` resources are available for the operation to complete.
`recv()`	EWOULDBLOCK	The socket is marked nonblocking, and the requested operation would block.

Refer to the man pages for the full list of error codes and descriptions.

IP Scripts

The Internet Protocol is the routing protocol in the TCP/IP stack responsible for addressing and delivery of packets. Versions include IPv4 and IPv6. See Figure 6-4, which illustrates where the IP layer resides relative to the network stack.

The IP layer is an ideal location for writing scripts with broad observability, because most common packets are processed by IP. The following are providers that can trace IP:

- **ip**: The stable IP provider (if available), for tracing send and receive events
- **mib**: For high-level statistics
- **fbt**: For tracing all kernel IP functions and arguments

If available, the stable ip provider can be used to write packet-oriented scripts. It currently provides probes for send, receive, and packet drop events. Listing the ip probes on Solaris Nevada, circa June 2010:

```
solaris# dtrace -ln ip:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
14352         ip                ip                ire_send_local_v6 receive
14353         ip                ip               ill_input_short_v6 receive
14354         ip                ip               ill_input_short_v4 receive
14355         ip                ip          ip_output_process_local receive
14356         ip                ip                ire_send_local_v4 receive
14381         ip                ip                   ip_drop_output drop-out
14382         ip                ip                    ip_drop_input drop-in
14435         ip                ip                ire_send_local_v6 send
14436         ip                ip          ip_output_process_local send
14437         ip                ip                ire_send_local_v4 send
14438         ip                ip                          ip_xmit send
```

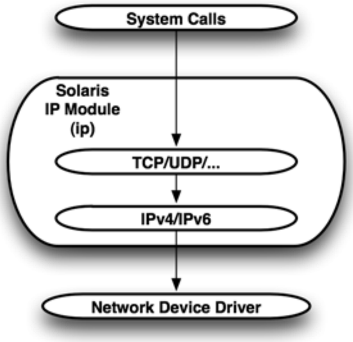


Figure 6-4 IP location in the Solaris network stack

To provide `ip:::send` and `ip:::receive` probes, nine different places in the kernel had to be traced (see the `FUNCTION` column). Without the ip provider, these nine places could be traced using the fbt provider; however, this would make for a fragile script because these functions can change between kernel versions. To illustrate this, consider the same listing of the ip provider on Solaris Nevada, circa December 2009:

```
solaris# dtrace -ln ip:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
30941         ip                ip                   ip_wput_local_v6 receive
30942         ip                ip                         ip_rput_v6 receive
30943         ip                ip                      ip_wput_local receive
30944         ip                ip                           ip_input receive
30961         ip                ip                     ip_inject_impl send
30962         ip                ip                           udp_xmit send
30963         ip                ip                   tcp_lsosend_data send
30964         ip                ip                      tcp_multisend send
30965         ip                ip                      tcp_send_data send
30966         ip                ip              ip_multicast_loopback send
30967         ip                ip                         ip_xmit_v6 send
30968         ip                ip                     ip_wput_ire_v6 send
30969         ip                ip                         ip_xmit_v4 send
30970         ip                ip                   ip_wput_ipsec_out send
30971         ip                ip                ip_wput_ipsec_out_v6 send
30972         ip                ip                       ip_wput_frag send
30973         ip                ip                   ip_wput_frag_mdt send
30974         ip                ip                        ip_wput_ire send
30975         ip                ip                    ip_fast_forward send
```

This version of Solaris Nevada needed to instrument 19 different locations for just the ip send and receive probes, and these function locations are very different.[14] Since the ip provider is the same, scripts based on it work on both versions. Scripts based on the fbt provider require substantial changes to keep functioning on different kernel versions. The `ipfbtsnoop.d` script is provided later as a demonstration of fbt tracing of IP.

The mib provider can be used for writing high-level statistics tools, which is demonstrated with the `ipstat.d` script.

The scripts in this section will demonstrate the mib, ip, and fbt providers.

fbt provider

Using the fbt provider is difficult, because it exposes the complexity and kernel implementation of the network stack, which may change from release to release. The TCP/IP stack source code is typically only the domain of kernel engineers or experienced users with knowledge of the kernel and the C programming language.

14. The reason is Erik Nordmark's IP Datapath Refactoring project (PSARC 2009/331), which reduced the number of ip functions in the Solaris TCP/IP stack, making the code much easier to follow and requiring fewer trace points for the ip provider.

The fbt-based scripts in this section were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available.

Solaris

To get an idea of the functions called, we will count IP probes that fire when sending 10,000 1KB messages on a recent version of Solaris Nevada, over TCP. The stable IP provider probe `ip:::send` will also be traced for comparison:

```
solaris# dtrace -n 'fbt::ip_*:entry { @[probefunc] = count(); }
 ip:::send { @["ip:::send"] = count(); }'
dtrace: description 'fbt::ip_*:entry ' matched 536 probes
^C

  ip_accept_tcp                                                  1
  ip_bind_connected_v4                                           1
  ip_bind_get_ire_v4                                             1
  ip_bind_laddr_v4                                               1
  ip_copymsg                                                     1
  ip_create_helper_stream                                        1
  ip_free_helper_stream                                          1
  ip_ire_advise                                                  1
  ip_massage_options                                             1
  ip_proto_bind_connected_v4                                     1
  ip_proto_bind_laddr_v4                                         1
  ip_squeue_get                                                  1
  ip_squeue_random                                               1
  ip_wput_attach_llhdr                                           1
  ip_wput_ioctl                                                  1
  ip_wput_ire                                                    1
  ip_wput_local                                                  1
  ip_xmit_v4                                                     1
  ip_get_numlifs                                                 2
  ip_ioctl_finish                                                2
  ip_process_ioctl                                               2
  ip_quiesce_conn                                                2
  ip_rput_process_broadcast                                      2
  ip_sioctl_get_lifnum                                           2
  ip_sioctl_copyin_setup                                         3
  ip_sioctl_lookup                                               7
  ip_wput_nondata                                                7
  ip_output                                                      8
  ip_output_options                                              8
  ip_input                                                    1801
  ip_tcp_input                                                5020
  ip:::send                                                  10025
  ip_cksum                                                   10025
  ip_ocsum                                                   10025
```

The `ip:::send` probe confirms that more than 10,000 packets were sent, but it isn't clear which function is sending the packets: We do not see an `ip_send` function that was called more than 10,000 times, for example. There is `ip_cksum()` and `ip_ocsum()`; however, they are for calculating checksums, not performing the send.

Trying the lower layers of GLDv3 (DLD, DLS, and MAC) yields the following:

```
solaris# dtrace -n 'fbt:dld::entry,fbt:dls::entry,fbt:mac::entry
{ @[probefunc] = count(); }'
dtrace: description 'fbt:dld::entry,fbt:dls::entry,fbt:mac::entry ' matched 303
probes
^C

  mac_soft_ring_intr_disable                                        3
  mac_soft_ring_intr_enable                                         3
  mac_soft_ring_poll                                                3
  dld_str_rx_unitdata                                               6
  str_unitdata_ind                                                  6
  mac_hwring_disable_intr                                          10
  mac_hwring_enable_intr                                           10
  mac_rx_ring                                                      10
  dls_accept                                                       15
  dls_accept_common                                                15
  i_dls_head_hold                                                  15
  i_dls_head_rele                                                  15
  i_dls_link_rx                                                    15
  i_dls_link_subchain                                              15
  mac_rx_deliver                                                   15
  dls_devnet_rele_tmp                                              16
  dls_devnet_stat_update                                           16
  dls_stat_update                                                  16
  mac_header_info                                                  21
  mac_vlan_header_info                                             21
  dld_wput                                                         23
  dls_header                                                       23
  mac_client_vid                                                   23
  mac_flow_get_desc                                                23
  mac_header                                                       23
  mac_sdu_get                                                      23
  proto_unitdata_req                                               23
  mac_stat_default                                                 24
  mac_stat_get                                                    320
  mac_rx_soft_ring_drain                                          467
  mac_soft_ring_worker_wakeup                                    1972
  mac_rx                                                         3439
  mac_rx_common                                                  3439
  mac_rx_soft_ring_process                                       3439
  mac_rx_srs_drain                                               3439
  mac_rx_srs_process                                             3439
  mac_rx_srs_proto_fanout                                        3439
  str_mdata_fastpath_put                                        10024
  mac_tx                                                        10047
```

Based on the counts and function names, this has identified two likely functions for the sending of ip packets: `mac_tx()` and `str_mdata_fastpath_put()`.

A little more investigation with DTrace shows the relationship between these functions:

```
solaris# dtrace -n 'fbt::mac_tx:entry { @[probefunc, stack()] = count(); }'
[...]
  mac_tx
            dld`str_mdata_fastpath_put+0xa4
```

```
              ip`tcp_send_data+0x94e
              ip`tcp_send+0xb69
              ip`tcp_wput_data+0x72c
              ip`tcp_output+0x830
              ip`squeue_enter+0x330
              ip`tcp_sendmsg+0xfd
              sockfs`so_sendmsg+0x1c7
              sockfs`socket_sendmsg+0x61
              sockfs`socket_vop_write+0x63
              genunix`fop_write+0xa4
              genunix`write+0x2e2
              genunix`write32+0x22
              unix`sys_syscall32+0x101
             5506
```

So, `str_mdata_fastpath_put()` calls `mac_tx()`. This also shows that `tcp_send_data()` calls the DLD layer directly, without calling ip functions. Shortcuts like this are not uncommon in the TCP/IP code to improve performance. It does make DTracing the functions a little confusing, as we saw when we were searching at the ip layer for the send function.

ip Provider Development

As we've just seen, the IP layer is skipped entirely on Solaris for this particular code path. If that's the case, where does the `ip:::send` probe fire from? A stack backtrace will show:

```
# dtrace -n 'ip:::send { @["ip:::send", stack(3)] = count(); }'
dtrace: description 'ip:::send ' matched 15 probes
[...]
  ip:::send
              ip`tcp_send+0xb69
              ip`tcp_wput_data+0x72c
              ip`tcp_rput_data+0x3342
             9988
```

It is firing from TCP, in the `tcp_send()` function. This led to consternation among kernel engineers during development: Should an IP probe fire at all, if the IP layer was skipped? Shouldn't we expose what really happens? Or, is the skipping of IP a Solaris kernel *implementation* detail, which is subject to change, and, which should be hidden from customers in a *stable* ip provider?

The implementation-detail argument won, and the `ip:::send` probe always fires, even if, to be technically accurate, the IP layer wasn't involved because of fastpath. This makes using the ip provider easier for end users (no need to worry about kernel implementation; read the RFCs instead) and allows the ip provider to be implemented on non-Solaris kernels such as Mac OS X and FreeBSD in the future.

Another way to learn the fbt probes is to map known mib events to the fbt functions, as demonstrated in the “mib Provider” section. And of course, if the source is available, it provides the best reference for the fbt probes and arguments.

Mac OS X

Here is the same 10,000 send packet experiment on Mac OS X:

```
solaris# dtrace -n 'fbt::ip_*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::ip_*:entry ' matched 23 probes
^C

  ip_savecontrol                                                    1
  ip_freemoptions                                                   6
  ip_ctloutput                                                      7
  ip_slowtimo                                                      30
  ip_input                                                       1102
  ip_output_list                                                 1160
  ip_randomid                                                    7132
```

ipstat.d

The `ipstat.d` script is covered in this section.

Script

This script retrieves IP statistics from five mib probes and sums their value in five separate aggregations. They are later printed on the same line. The mib statistics were chosen because they looked interesting and useful; this can be customized by adding more of the available mib statistics as desired.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        LINES = 20; line = 0;
}

profile:::tick-1sec
/--line <= 0/
{
        printf("  IP IF:  %12s %12s %12s %12s %12s\n", "out(bytes)",
            "outDiscards", "in(bytes)", "inDiscards", "inErrors");
        line = LINES;
}

mib:::ipIfStatsHCInOctets       { @in = sum(arg0);      }
mib:::ipIfStatsHCOutOctets      { @out = sum(arg0);     }
mib:::ipIfStatsInDiscards       { @inDis = sum(arg0);   }
mib:::ipIfStatsOutDiscards      { @outDis = sum(arg0);  }
mib:::ipIfStatsIn*Errors        { @inErr = sum(arg0);   }

```

```
24  profile:::tick-1sec
25  {
26          printa("          %@12d %@12d %@12d %@12d %@12d\n",
27              @out, @outDis, @in, @inDis, @inErr);
28          clear(@out); clear(@outDis); clear(@in); clear(@inDis); clear(@inErr);
29  }
```

Script ipstat.d

A variable called `line` is used to track when to reprint the header. This happens every 20 lines; without it, the screen could fill with numbers and become difficult to follow.

Line 29 uses a multiple aggregation `printa()` to generate the output. If none of those aggregations contained data at this point, no output would be generated because `printa()` skips printing when all of its aggregations arguments are empty. Once some IP events have occurred, the aggregations are cleared on line 28—and not truncated—so that they still contain data (albeit zero), which ensures that `printa()` will print something out (and then continue to do so every second), even if that is entirely zeros.

Example

`ipstat.d` was executed on a system that was receiving a large TCP transfer:

```
solaris# ipstat.d
  IP IF:    out(bytes)  outDiscards    in(bytes)   inDiscards     inErrors
                 41880            0     12153018            0            0
                 40514            0     11676695            0            0
                 36840            0     10670889            0            0
                 46720            0     11853477            0            0
                 45676            0     10768995            0            0
                 46068            0      9895095            0            0
                 63920            0     11829585            0            0
                 46560            0      7968817            0            0
                 79720            0     11850263            0            0
                227556            1      9475738            0            0
                 80000            0     11901382            0            0
[...]
```

The `outDiscards` error was unexpected and prompts further investigation with DTrace (providing the error is repeatable), such as observing the kernel stack trace when that probe fired.

ipio.d

Trace IPv4 and IPv6 send and receive events using the ip provider (if available). On Solaris systems with the ip provider, this script is available in `/usr/demo/dtrace`.

Script

This is a simple script to print out data from the ip provider and could be the starting point for more sophisticated scripts.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf(" %3s %10s %15s    %15s %8s %6s\n", "CPU", "DELTA(us)",
            "SOURCE", "DEST", "INT", "BYTES");
        last = timestamp;
}

ip:::send
{
        this->delta = (timestamp - last) / 1000;
        printf(" %3d %10d %15s -> %15s %8s %6d\n", cpu, this->delta,
            args[2]->ip_saddr, args[2]->ip_daddr, args[3]->if_name,
            args[2]->ip_plength);
        last = timestamp;
}

ip:::receive
{
        this->delta = (timestamp - last) / 1000;
        printf(" %3d %10d %15s <- %15s %8s %6d\n", cpu, this->delta,
            args[2]->ip_daddr, args[2]->ip_saddr, args[3]->if_name,
            args[2]->ip_plength);
        last = timestamp;
}
```

Script ipio.d

The CPU ID is printed as a clue that DTrace may shuffle output on multi-CPU systems. If this becomes a problem, print a time stamp and post-process, sorting on the time value.

The delta time calculation (for `this->delta`) is simple: the time since the last event, which is kept in the `last` scalar global variable.

Example

This example output shows tracing packets as they pass in and out of tunnels:

```
# ipio.d
 CPU  DELTA(us)          SOURCE                DEST       INT  BYTES
   1     598913    10.1.100.123 ->   192.168.10.75  ip.tun0     68
   1         73   192.168.1.108 ->     192.168.5.1     nge0    140
   1      18325   192.168.1.108 <-     192.168.5.1     nge0    140
   1         69    10.1.100.123 <-   192.168.10.75  ip.tun0     68
   0     102921    10.1.100.123 ->   192.168.10.75  ip.tun0     20
   0         79   192.168.1.108 ->     192.168.5.1     nge0     92
```

Note that the delta time between output lines is printed. These may not necessarily be related. They could be for different sessions; they may also become difficult to read if DTrace shuffles the output (it's unclear what the 102921 us time refers to). Even if they look likely to be related (lines 2 and 3, with a delta of 18325 us), they could be for two packets between the same hosts that happened to be in flight, not necessarily a round-trip time (RTT) measurement. To measure RTT, examine sequence numbers at the TCP layer.

ipproto.d

The `ipproto.d` script summarizes IP traffic by the next-level protocol and packet count and uses the ip provider. This is a simple but useful high-level view of IP activity; anything suspicious can be examined more deeply with additional DTrace.

Script

This script is very simple, aggregating events and then printing them. This is the intent of stable providers: to allow scripting to be easy and concise.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           printf("Tracing... Hit Ctrl-C to end.\n");
8   }
9
10  ip:::send,
11  ip:::receive
12  {
13          this->protostr = args[2]->ip_ver == 4 ?
14              args[4]->ipv4_protostr : args[5]->ipv6_nextstr;
15          @num[args[2]->ip_saddr, args[2]->ip_daddr, this->protostr] = count();
16  }
17
18  dtrace:::END
19  {
20          printf("   %-28s %-28s %6s %8s\n", "SADDR", "DADDR", "PROTO", "COUNT");
21          printa("   %-28s %-28s %6s %@8d\n", @num);
22  }
```

Script ipproto.d

On line 13 the IP version was checked to determine where to read the next-level protocol from (IPv4 or IPv6 header).

Example

This example shows a variety of protocols and address. The hosts 192.168.1.108 and 192.168.1.109 were busy transferring packets in a TCP session:

```
solaris# ipproto.d
Tracing... Hit Ctrl-C to end.
^C
SADDR                          DADDR                          PROTO    COUNT
192.168.1.108                  192.168.155.32                   UDP        1
192.168.1.108                  192.168.17.55                    UDP        1
192.168.1.108                  192.168.228.54                   UDP        1
192.168.1.108                  192.168.1.5                      UDP        1
192.168.1.108                  192.168.2.27                    ICMP        1
192.168.1.200                  192.168.3.255                    UDP        1
192.168.1.5                    192.168.1.108                    UDP        1
192.168.2.27                   192.168.1.108                   ICMP        1
fe80::214:4fff:fe3b:76c8       ff02::1                       ICMPV6        1
fe80::2e0:81ff:fe5e:8308       fe80::214:4fff:fe3b:76c8      ICMPV6        1
fe80::2e0:81ff:fe5e:8308       ff02::1:2                        UDP        1
192.168.1.185                  192.168.1.255                    UDP        2
192.168.1.211                  192.168.1.255                    UDP        3
192.168.1.109                  192.168.1.108                    TCP      428
192.168.1.108                  192.168.1.109                    TCP      789
```

ipfbtsnoop.d

The previous examples used stable providers such as ip, which may not be available on your operating system. To demonstrate what is possible without these stable providers, the `ipfbtsnoop.d` script was written for Solaris using the unstable fbt provider. It also avoids using DTrace convenience functions, which may also not be available either, such as `inet_ntoa()` and `ntohs()`.

This script is a demonstration of fbt tracing of IP, not as a script that is expected to work anywhere. Since it hooks into the IP implementation, it is extremely brittle and is expected to not work on most Solaris versions. (Depending on the extent of kernel differences, some Solaris versions may only require minor updates for this script to work.)

Script

The `-C` option is used with DTrace to run the preprocessor. This allows macros to be defined that can be reused: Here `IPV4_ADDR_TO_STR()` and `BSWAP_16()` were defined on the assumption that the DTrace functions `inet_ntoa()` and `ntohs()` may not be available (they weren't on very first releases of DTrace on Solaris[15]), and, that this is a little-endian system[16] (otherwise, `ntohs()` / `BSWAP_16()` are not needed). This is just a demonstration of one way to achieve this; possible improvements include checking endian-ness programatically with the preprocessor to check whether using `BSWAP_16()` is necessary (as demonstrated in the `tcpsnoop_snv.d`

15. `ntohs()` was added in CR 6282214, "Byte Ordering Functions in libdtrace."; `inet_ntoa()` was added in CR 6558517, "need DTrace versions of IP address to string functions, like inet_ntop()."

16. x86 systems are little-endian; SPARC is big-endian.

script) and, using a `#include` statement to include `sys/byteorder.h`, to avoid needing to define `BSWAP_16()` in the script.

This script only traces IPv4 traffic. It could be enhanced to handle IPv6 as well.

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option switchrate=10hz

#define ETHERTYPE_IP            (0x0800)        /* IP protocol */
#define ETHERTYPE_IPV6          (0x86dd)        /* IPv6 */

#define IPPROTO_IP              0
#define IPPROTO_ICMP            1
#define IPPROTO_IGMP            2
#define IPPROTO_TCP             6
#define IPPROTO_UDP             17

#define DL_ETHER                0x4

#define IPH_HDR_VERSION(ipha) \
        ((int)(((ipha_t *)ipha)->ipha_version_and_hdr_length) >> 4)

/* stringify an IPv4 address without inet*() being available */
#define IPV4_ADDR_TO_STR(string, addr)                                  \
        this->a = (uint8_t *)&addr;                                     \
        this->addr1 = strjoin(lltostr(this->a[0] + 0ULL), strjoin(".",  \
            strjoin(lltostr(this->a[1] + 0ULL), ".")));                 \
        this->addr2 = strjoin(lltostr(this->a[2] + 0ULL), strjoin(".",  \
            lltostr(this->a[3] + 0ULL)));                               \
        string = strjoin(this->addr1, this->addr2);

/* convert net to host byte order for little-endian systems  */
#define BSWAP_16(host, net)                                             \
        host = (net & 0xFF00) >> 8;                                     \
        host |= (net & 0xFF) << 8;

dtrace:::BEGIN
{
        /* selected protocols; see /usr/include/netinet/in.h for full list */
        ipproto[IPPROTO_IP] = "IP";
        ipproto[IPPROTO_ICMP] = "ICMP";
        ipproto[IPPROTO_IGMP] = "IGMP";
        ipproto[IPPROTO_TCP] = "TCP";
        ipproto[IPPROTO_UDP] = "UDP";

        printf("%-15s %-8s %-8s %-15s   %-15s %5s %5s\n", "TIME(us)",
            "ONCPU", "INT", "SOURCE", "DEST", "BYTES", "PROTO");
}

fbt::ip_input:entry
{
        this->mp = args[2];
        this->ill = args[0];
        this->ipha = (ipha_t *)this->mp->b_rptr;
        this->name = stringof(this->ill->ill_name);
        this->ok = 1;
}

/* rewrite for dls_tx() on older Solaris kernels */
fbt::mac_tx:entry
```

continues

```
58  {
59          this->mc = (mac_client_impl_t *)args[0];
60  }
61
62  /* filter out non-Ethernet calls */
63  fbt::mac_tx:entry
64  /this->mc->mci_mip->mi_info.mi_nativemedia == DL_ETHER/
65  {
66          this->mp = args[1];
67          this->eth = (struct ether_header *)this->mp->b_rptr;
68          this->type = this->eth->ether_type;
69  }
70
71  /* filter out non-IP calls */
72  fbt::mac_tx:entry
73  /this->type == ETHERTYPE_IP || this->type == ETHERTYPE_IPV6/
74  {
75          this->ipha = (ipha_t *)&this->mp->b_rptr[sizeof (struct ether_header)];
76          this->name = this->mc->mci_name;
77          this->ok = 1;
78  }
79
80  fbt::ip_input:entry, fbt::mac_tx:entry
81  /this->ok && IPH_HDR_VERSION(this->ipha) == 4/
82  {
83          BSWAP_16(this->pktlen, this->ipha->ipha_length);
84          IPV4_ADDR_TO_STR(this->src, this->ipha->ipha_src);
85          IPV4_ADDR_TO_STR(this->dst, this->ipha->ipha_dst);
86
87          this->proto = ipproto[this->ipha->ipha_protocol] != NULL ?
88              ipproto[this->ipha->ipha_protocol] :
89              lltostr(this->ipha->ipha_protocol);
90
91          printf("%-15d %-8.8s %-8.8s %-15s > %-15s %5d %5s\n",
92              timestamp / 1000, execname, this->name, this->src, this->dst,
93              this->pktlen, this->proto);
94  }
```

Script ipfbtsnoop.d

Various constants are defined on lines 6 to 15 to highlight what is used in the remainder of the script. These constants can be included from their respective header files instead, making the script a little more robust (in case of changes to those values).

Example

The `ipfbtsnoop.d` script was executed for a short period:

```
solaris# ipfbtsnoop.d
TIME(us)        ONCPU    INT      SOURCE            DEST              BYTES PROTO
75612897006     sched    nge0     192.168.1.109   > 192.168.2.145       40   TCP
75612904644     sched    nge0     192.168.2.53    > 192.168.2.145       84  ICMP
75612904726     sched    nge0     192.168.2.145   > 192.168.2.53        84  ICMP
75612944405     sched    nge0     192.168.1.109   > 192.168.2.145       40   TCP
75613054289     sched    nxge5    0.0.0.0         > 255.255.255.255    328   UDP
75613054200     sched    nge0     0.0.0.0         > 255.255.255.255    328   UDP
75613084667     sched    nge0     192.168.2.53    > 192.168.2.145       88   TCP
```

```
75613097038     sched    nge0     192.168.1.109   > 192.168.2.145      40   TCP
75613054265     sched    nxge1    0.0.0.0         > 255.255.255.255   328   UDP
75613084666     sched    nxge1    192.168.100.4   > 192.168.100.50     88   TCP
75613084779     sched    nxge1    192.168.100.4   > 192.168.100.50     88   TCP
75613144674     sched    nxge1    192.168.100.4   > 192.168.100.50     40   TCP
75613144421     sched    nge0     192.168.1.109   > 192.168.2.145      40   TCP
75613144631     sched    nge0     192.168.2.53    > 192.168.2.145      40   TCP
^C
```

To trace inbound and outbound IP packets, the `ip_input()` and `mac_tx()` functions were traced, which seems to work. However, it is likely that certain packet types will not be traced using these two functions alone, based on the following observation from the stable ip provider:

```
# dtrace -n 'ip:::receive { @ = count(); }'
dtrace: description 'ip:::receive ' matched 4 probes
[...]
```

To trace `ip:::receive`, DTrace has had to enable four instances of the `ip:::send` probe. To show where they are placed, run this:

```
# dtrace -ln 'ip:::receive'
   ID   PROVIDER            MODULE                          FUNCTION NAME
30941         ip                ip                   ip_wput_local_v6 receive
30942         ip                ip                         ip_rput_v6 receive
30943         ip                ip                      ip_wput_local receive
30944         ip                ip                           ip_input receive
```

Our script traces `ip_input()`, but we missed `ip_wput_local()` and the IPv6 functions.

TCP Scripts

The Transmission Control Protocol (RFC 793) is a reliable transmission protocol and part of the TCP/IP stack and is shown in Figure 6-5.

On both client and server, use DTrace to answer the following.

How many outbound connections were established? By client, port?

How many inbound connections were accepted? By client, port?

How long did TCP connections take?

How much data was sent? I/O size? By client, port?

What was the round trip time? Average? Maximum? By client/destination?

How long were connections established? What was the average throughput?

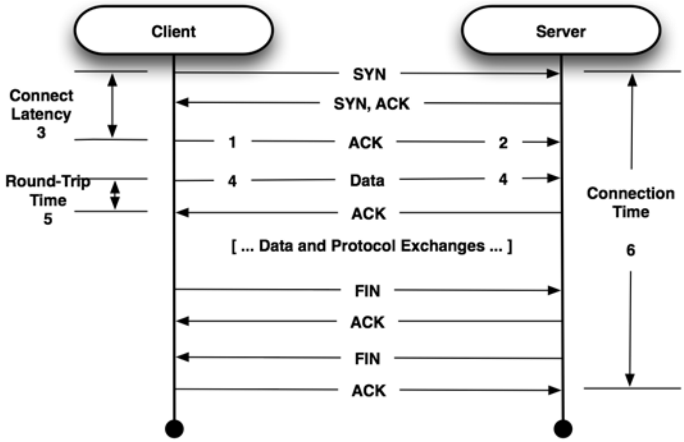


Figure 6-5 TCP handshake and I/O

TCP scripts can be written using the tcp provider (if available) for TCP events and/or the `mib` and `syscall` providers for an overall idea of TCP usage across the system. To examine the internal operation of the TCP layer in the network stack, the unstable fbt provider can be used, with the same caveats as fbt tracing of IP (as discussed for `ipfbtsnoop.d`).

Figure 6-5 shows a typical TCP session between a client and a server, along with questions to consider, such as counting outbound connections. It should be noted that the client-server and outbound-inbound terminology refer to a common model for using TCP, but it's not the only model. TCP connections can be local, for example, over the loopback interface, so the terms *inbound* and *outbound* lose meaning. The terms *client* and *server* may also be meaningless, depending on the type of TCP connection. The terms used by TCP specification (RFC793) are *active* and *passive*, which typically refer to the client and server ends, respectively.

The TCP provider uses the probe name `connect-established` to refer to a TCP active open (for example, a client connects to a server) and the probe name `accept-established` to refer to a TCP passive open (for example, a server accepts a client connection).

The scripts shown in this section demonstrate high-level TCP observability and can be the starting point for more complex TCP scripts, such as scripts to examine TCP congestion, window size changes, and so on.

tcp Provider

Listing probes from the tcp provider (Solaris Nevada, circa June 2010) yields the following:

```
solaris# dtrace -ln tcp:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
14143        tcp                ip                    tcp_input_data connect-refused
14153        tcp                ip                    tcp_input_data accept-established
14155        tcp                ip                    tcp_input_data connect-
established
14174        tcp                ip                      tcp_xmit_ctl accept-refused
14220        tcp                ip                    tcp_input_data receive
14221        tcp                ip                tcp_input_listener receive
14222        tcp                ip          tcp_xmit_listeners_reset receive
14223        tcp                ip                   tcp_fuse_output receive
14224        tcp                ip                tcp_input_listener send
14225        tcp                ip                     tcp_ss_rexmit send
14226        tcp                ip                   tcp_sack_rexmit send
14227        tcp                ip              tcp_xmit_early_reset send
14228        tcp                ip                      tcp_xmit_ctl send
14229        tcp                ip                      tcp_xmit_end send
14230        tcp                ip                          tcp_send send
14231        tcp                ip                     tcp_send_data send
14232        tcp                ip                        tcp_output send
14233        tcp                ip                   tcp_fuse_output send
14250        tcp                ip                    tcp_do_connect connect-request
14269        tcp                ip                         tcp_bindi state-change
14270        tcp                ip                    tcp_input_data state-change
14271        tcp                ip                tcp_input_listener state-change
14272        tcp                ip                       tcp_xmit_mp state-change
14273        tcp                ip                     tcp_do_listen state-change
14274        tcp                ip                    tcp_do_connect state-change
14275        tcp                ip                     tcp_do_unbind state-change
14276        tcp                ip                        tcp_reinit state-change
14277        tcp                ip             tcp_disconnect_common state-change
14278        tcp                ip                  tcp_closei_local state-change
14279        tcp                ip                   tcp_clean_death state-change
```

This TCP provider version traces sends and receives, connections, and TCP state changes. The send and receive probes trace I/O at the TCP layer. For convenience, this chapter will sometimes refer to this as tracing TCP *packets*; however, technically they may not map one-to-one to packets as seen on the wire: For example, IP will fragment large packets into MTU-sized packets (or return an ICMP error).

The tcp provider is one of the newest (integrated into Solaris Nevada build 142) and may not yet be available for your operating system version. If not, these tcp provider-based scripts still serve as examples of what TCP data can be useful to retrieve and could (with some effort) be reimplemented as fbt provider-based scripts until the tcp provider is available.

fbt Provider

Using the fbt provider is difficult, because it exposes the complexity and kernel implementation of the network stack, which may change from release to release. The TCP/IP stack source code is typically only the domain of kernel engineers or experienced users with knowledge of the kernel and the C programming language.

Listing the fbt probes available for tcp functions on Solaris Nevada, circa December 2009 (we deliberately switched to an older version before the tcp provider was available, where fbt was the only option apart from the mib provider):

```
solaris# dtrace -ln 'fbt::tcp_*:'
   ID   PROVIDER            MODULE                          FUNCTION NAME
56671        fbt                ip              tcp_conn_constructor entry
56672        fbt                ip              tcp_conn_constructor return
56673        fbt                ip               tcp_conn_destructor entry
56674        fbt                ip               tcp_conn_destructor return
56902        fbt                ip                  tcp_set_ws_value entry
56903        fbt                ip                  tcp_set_ws_value return
56904        fbt                ip              tcp_time_wait_remove entry
56905        fbt                ip              tcp_time_wait_remove return
56906        fbt                ip              tcp_time_wait_append entry
56907        fbt                ip              tcp_time_wait_append return
56908        fbt                ip                tcp_close_detached entry
56909        fbt                ip                tcp_close_detached return
56910        fbt                ip              tcp_bind_hash_remove entry
56911        fbt                ip              tcp_bind_hash_remove return
56912        fbt                ip                        tcp_accept entry
56913        fbt                ip                        tcp_accept return
[...truncated...]
```

On this version of the kernel, 504 probes were listed for tracing the internals of TCP. The number will change with kernel updates to match the current kernel implementation.

To get an idea of the tcp functions called, we'll count probes that fire when sending 10,000 1KB messages over TCP:

```
solaris# dtrace -n 'fbt::tcp_*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::tcp_*:entry ' matched 252 probes
^C

  tcp_acceptor_hash_remove                                          1
  tcp_adapt_ire                                                     1
  tcp_bind                                                          1
[...truncated...]
  tcp_timeout                                                     140
  tcp_clrqfull                                                    157
  tcp_setqfull                                                    157
  tcp_send                                                       4532
  tcp_set_rto                                                    5093
  tcp_parse_options                                              5132
  tcp_rput_data                                                  5152
  tcp_fill_header                                                9998
  tcp_output                                                    10109
  tcp_wput                                                      10111
  tcp_send_data                                                 10151
  tcp_send_find_ire                                             10151
  tcp_send_find_ire_ill                                         10151
  tcp_wput_data                                                 12630
```

The functions with higher counts (in the 10,000s) are likely to be those processing I/O, and those with lower counts (less than 10) are those that initiate the connection. Perform this experiment in the opposite direction (or trace on the remote host) to see the TCP receive side.

The relationship between these functions can be illustrated by examining stack traces, as shown in the "fbt Provider" section. Another way to learn the fbt probes is to map known mib events to the fbt functions, as demonstrated in the "mib Provider" section. And of course, if the source is available, it provides the best reference for the fbt probes and arguments.

The fbt-based scripts later in this section were based on OpenSolaris circa December 2009 and may not work on other OSs and releases without changes. Even if these scripts no longer execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available.

tcpstat.d

The `tcpstat.d` is an example of using the mib provider to track statistics for a specific protocol.

Script

Various TCP statistics are traced from the mib provider on Solaris and printed every second, in a similar fashion to the `ipstat.d` script:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN                  { LINES = 20; line = 0; }

profile:::tick-1sec
/--line <= 0/
{
        printf("  TCP bytes:  %6s %12s %12s %12s %12s\n",
            "out", "outRetrans", "in", "inDup", "inUnorder");
        line = LINES;
}
mib:::tcpOutDataBytes, mib:::tcpRetransBytes, mib:::tcpInDataInorderBytes,
mib:::tcpInDataDupBytes, mib:::tcpInDataUnorderBytes
{
        /* some of these probes can return -1 */
        this->bytes = (int)arg0 > 0 ? arg0 : 0;
}

mib:::tcpOutDataBytes           { @out = sum(this->bytes);   }
mib:::tcpRetransBytes           { @outRe = sum(this->bytes); }
mib:::tcpInDataInorderBytes     { @in = sum(this->bytes);    }
mib:::tcpInDataDupBytes         { @inDup = sum(this->bytes); }
mib:::tcpInDataUnorderBytes     { @inUn = sum(this->bytes);  }
```

continues

```
26
27  profile:::tick-1sec
28  {
29          printa("        %@12d %@12d %@12d %@12d %@12d\n",
30              @out, @outRe, @in, @inDup, @inUn);
31          clear(@out); clear(@outRe); clear(@in); clear(@inDup); clear(@inUn);
32  }
```

Script tcpstat.d

A variable called `line` is used to track when to reprint the header. This happens every 20 lines; without it, the screen could fill with numbers and become difficult to follow.

Line 29 uses a multiple aggregation `printa()` to generate the output. If none of those aggregations contained data at this point, no output will be generated because `printa()` skips printing when all of its aggregations arguments are empty. Once some TCP events have occurred, the aggregations are cleared on line 31—and not truncated—so that they still contain data (albeit zero), which ensures that `printa()` will print something out (and then continue to do so every second), even if that is entirely zeros.

Example

This example output shows steady, TCP-inbound data after the third line of output.

```
solaris# tcpstat.d
  TCP bytes:      out   outRetrans          in       inDup    inUnorder
                18100            0       19941           0            0
                16812            0       21440           0            0
                16752            0     3260812           0            0
                16946            0    11605173           0            0
                16704            0    11358911           0            0
                16812            0    10718226           0            0
                17400            0    11500106           0            0
                17864            0    11459260           0            0
                16704            0    11460956           0            0
[...]
```

tcpaccept.d

`tcpaccept.d` summarizes which clients have established connections to which TCP ports, using the tcp provider.

Script

The script is basically a one-liner with output formatting, again illustrating the point of stable providers, to allow powerful scripts to be written simply:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

tcp:::accept-established
{
        @num[args[2]->ip_saddr, args[4]->tcp_dport] = count();
}

dtrace:::END
{
        printf("   %-26s %-8s %8s\n", "HOST", "PORT", "COUNT");
        printa("   %-26s %-8d %@8d\n", @num);
}
```

Script tcpaccept.d

The tcp provider has the source IP address available as the string `args[2]->ip_saddr`, which can contain either IPv4 or IPv6 address strings. For the accept-established probe, `args[3]->tcps_raddr` would also work because it is the remote address string.

Example

Several inbound TCP connections were established as the `tcpaccept.d` script was running:

```
solaris# tcpaccept.d
Tracing... Hit Ctrl-C to end.
^C
   HOSTNAME                     PORT        COUNT
   192.168.1.109                23              1
   192.168.1.109                80              1
   fe80::214:4fff:fe3b:76c8     22              1
   192.168.1.109                22              3
   192.168.1.109                61360           6
```

This shows that a single client, 192.168.1.109, was responsible for most of the connections. It made three connections to port 22 (`ssh`) and six to port 61360 (an RPC port). An IPv6 client, fe80::214:4fff:fe3b:76c8, performed one connection to port 22 (`ssh`).

tcpacceptx.d

This is the same as `tcpaccept.d` but has been enhanced to use extra formatting characters that are not yet available in most versions of DTrace.[17] The characters are as follows:

- `%I` Resolve IP addresses to host names
- `%P` Resolve ports to names

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

tcp:::accept-established
{
        @num[args[2]->ip_saddr, args[4]->tcp_dport] = count();
}

dtrace:::END
{
        printf("   %-26s %-8s %8s\n", "HOSTNAME", "PORT", "COUNT");
        printa("   %-26I %-8P %@8d\n", @num);
}
```

Script tcpacceptx.d

Example

This time the `tcpacceptx.d` script shows the fully qualified host names and port names for inbound TCP connections:

```
solaris# tcpacceptx.d
Tracing... Hit Ctrl-C to end.
^C
   HOSTNAME                   PORT        COUNT
   deimos.sf.fishworks.com    telnet          1
   deimos.sf.fishworks.com    http            1
   phobos6.sf.fishworks.com   ssh             1
   deimos.sf.fishworks.com    ssh             3
   deimos.sf.fishworks.com    61360           7
```

17. These are currently only implemented on the Oracle Sun ZFS Storage 7000 series, which at times has implemented features before they are integrated into mainstream OpenSolaris and Solaris.

phobos6 is a host name for an IPv6 address. Port 61360 wasn't translated; an investigation found that it was dynamically allocated for RPC:

```
solaris# rpcinfo -p | grep 61360
    100005    1   tcp  61360  mountd
    100005    2   tcp  61360  mountd
    100005    3   tcp  61360  mountd
```

It was for mountd, NFS mounts.

tcpconnect.d

The tcpaccept.d scripts traced inbound TCP connections. tcpconnect.d traces outbound TCP connections.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

tcp:::connect-established
{
        @num[args[2]->ip_daddr, args[4]->tcp_dport] = count();
}

dtrace:::END
{
        printf("   %-26s %-8s %8s\n", "HOST", "PORT", "COUNT");
        printa("   %-26s %-8d %@8d\n", @num);
}
```

Script tcpconnect.d

The tcp provider has the destination IP address available as the string args[2]->ip_daddr, which can contain both IPv4 and IPv6 address strings. For the connect-established probe, args[3]->tcps_raddr would also work because it's the remote address string.

Example

Two outbound TCP connections were made to 72.5.124.61 port 80.

```
solaris# tcpconnect.d
Tracing... Hit Ctrl-C to end.
^C
   HOST                           PORT        COUNT
   192.168.1.109                  22              1
   72.5.124.61                    80              2
```

tcpioshort.d

This is a short version of the `tcpio.d` script to demonstrate the basics of the tcp provider. It traces TCP sends and receives, with source and destination addresses, the port, and IP payload bytes.

Script

```
#!/usr/sbin/dtrace -s

tcp:::send, tcp:::receive
{
        printf("%15s:%-5d  ->  %15s:%-5d %d bytes",
            args[2]->ip_saddr, args[4]->tcp_sport,
            args[2]->ip_daddr, args[4]->tcp_dport,
            args[2]->ip_plength);
}
```

Script tcpioshort.d

Example

This is a quick way to identify TCP traffic. The `tcpio.d` script traces the same probes but formats neatly. The output to `tcpioshort.d` can scroll quickly because this is running an `ssh` session, and it is tracing TCP events caused by itself printing output, which is a feedback loop.

```
solaris# tcpioshort.d
dtrace: script './tcpioshort.d' matched 8 probes
CPU     ID           FUNCTION:NAME
  0 31437    tcp_send_data:send  192.168.2.145:2049  ->    192.168.2.8:1021  100 bytes
  6 31079 tcp_rput_data:receive  192.168.100.4:44091 -> 192.168.100.50:3260  20 bytes
  6 31437    tcp_send_data:send  192.168.2.145:215   ->  192.168.1.109:54575 20 bytes
  8 31079 tcp_rput_data:receive   192.168.2.53:36395 ->  192.168.2.145:3260  68 bytes
  8 31079 tcp_rput_data:receive   192.168.2.53:36395 ->  192.168.2.145:3260  20 bytes
  8 31079 tcp_rput_data:receive  192.168.1.109:54575 ->  192.168.2.145:215   617 bytes
  8 31079 tcp_rput_data:receive  192.168.1.109:54575 ->  192.168.2.145:215   201 bytes
  8 31079 tcp_rput_data:receive  192.168.1.109:54575 ->  192.168.2.145:215   20 bytes
  8 31079 tcp_rput_data:receive    192.168.2.8:1021  ->  192.168.2.145:2049  260 bytes
  8 31079 tcp_rput_data:receive    192.168.2.8:1021  ->  192.168.2.145:2049  252 bytes
 11 31437    tcp_send_data:send  192.168.2.145:3260  ->   192.168.2.53:36395 68 bytes
 11 31437    tcp_send_data:send 192.168.100.50:3260  ->  192.168.100.4:44091 68 bytes
 12 31437    tcp_send_data:send  192.168.2.145:22    ->  192.168.1.109:36683 100 bytes
 12 31437    tcp_send_data:send  192.168.2.145:22    ->  192.168.1.109:36683 164 bytes
 12 31437    tcp_send_data:send  192.168.2.145:22    ->  192.168.1.109:36683 164 bytes
[...]
```

tcpio.d

The `tpcio.d` script traces tcp send and receives, showing various details from the TCP and IP headers formatted into columns.

Script

Rather than printing the IP payload bytes that would include the TCP header, lines 14 and 22 of this script calculate the actual TCP payload bytes (which `tcpioshort.d` did, including the length of the TCP header). Lines 31 to 40 print TCP flags at the end of the line; 40 prints a backspace (`\b`) to move the cursor back over any extra pipe (`|`) character (which is then overwritten using the `)` character).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8           printf("%-3s %15s:%-5s      %15s:%-5s %6s %s\n", "CPU",
9               "LADDR", "LPORT", "RADDR", "RPORT", "BYTES", "FLAGS");
10  }
11
12  tcp:::send
13  {
14          this->length = args[2]->ip_plength - args[4]->tcp_offset;
15          printf("%-3d %15s:%-5d  ->  %15s:%-5d %6d (", cpu,
16              args[2]->ip_saddr, args[4]->tcp_sport,
17              args[2]->ip_daddr, args[4]->tcp_dport, this->length);
18  }
19
20  tcp:::receive
21  {
22          this->length = args[2]->ip_plength - args[4]->tcp_offset;
23          printf("%-3d %15s:%-5d  <-  %15s:%-5d %6d (", cpu,
24              args[2]->ip_daddr, args[4]->tcp_dport,
25              args[2]->ip_saddr, args[4]->tcp_sport, this->length);
26  }
27
28  tcp:::send,
29  tcp:::receive
30  {
31          printf("%s", args[4]->tcp_flags & TH_FIN ? "FIN|" : "");
32          printf("%s", args[4]->tcp_flags & TH_SYN ? "SYN|" : "");
33          printf("%s", args[4]->tcp_flags & TH_RST ? "RST|" : "");
34          printf("%s", args[4]->tcp_flags & TH_PUSH ? "PUSH|" : "");
35          printf("%s", args[4]->tcp_flags & TH_ACK ? "ACK|" : "");
36          printf("%s", args[4]->tcp_flags & TH_URG ? "URG|" : "");
37          printf("%s", args[4]->tcp_flags & TH_ECE ? "ECE|" : "");
38          printf("%s", args[4]->tcp_flags & TH_CWR ? "CWR|" : "");
39          printf("%s", args[4]->tcp_flags == 0 ? "null " : "");
40          printf("\b)\n");
41  }
```

Script tcpio.d

Examples

Several examples follow including tracing a TCP handshake, port closed, and loopback traffic.

Tracing a TCP Handshake. The output includes a FLAGS column for TCP flags, allowing TCP state to be inferred:

```
solaris# tcpio.d
CPU           LADDR:LPORT                 RADDR:RPORT  BYTES FLAGS
13     192.168.2.145:22      ->    192.168.1.109:36683    112 (PUSH|ACK)
8      192.168.2.145:22      <-    192.168.1.109:36683      0 (ACK)
13     192.168.2.145:22      ->    192.168.1.109:36683    112 (PUSH|ACK)
6      192.168.2.145:215     ->    192.168.1.109:54340      0 (ACK)
8      192.168.2.145:22      <-    192.168.1.109:36683      0 (ACK)
8      192.168.2.145:215     <-    192.168.1.109:54340    597 (PUSH|ACK)
8      192.168.2.145:215     <-    192.168.1.109:54340    181 (PUSH|ACK)
9      192.168.2.145:215     ->    192.168.1.109:54340    500 (PUSH|ACK)
8      192.168.2.145:55190   ->      192.168.1.3:22         0 (SYN)
8      192.168.2.145:55190   <-      192.168.1.3:22         0 (SYN|ACK)
8      192.168.2.145:55190   ->      192.168.1.3:22         0 (ACK)
4      192.168.2.145:55190   ->      192.168.1.3:22        20 (PUSH|ACK)
4      192.168.2.145:55190   ->      192.168.1.3:22       504 (PUSH|ACK)
8      192.168.2.145:55190   <-      192.168.1.3:22        20 (PUSH|ACK)
8      192.168.2.145:55190   ->      192.168.1.3:22         0 (ACK)
8      192.168.2.145:55190   <-      192.168.1.3:22         0 (ACK)
8      192.168.2.145:55190   <-      192.168.1.3:22         0 (ACK)
8      192.168.2.145:215     <-    192.168.1.109:33837    597 (PUSH|ACK)
6      192.168.2.145:215     ->    192.168.1.109:33837      0 (ACK)
8      192.168.2.145:215     <-    192.168.1.109:33837    181 (PUSH|ACK)
8      192.168.2.145:215     <-    192.168.1.109:33837      0 (ACK)
13     192.168.2.145:215     ->    192.168.1.109:33837    500 (PUSH|ACK)
4      192.168.2.145:55190   ->      192.168.1.3:22        24 (PUSH|ACK)
8      192.168.2.145:55190   <-      192.168.1.3:22       376 (PUSH|ACK)
8      192.168.2.145:55190   ->      192.168.1.3:22         0 (ACK)
[...]
```

The output includes an outbound TCP connection to 192.168.1.3 port 22 (SSH); the TCP handshake is visible in the FLAGS column: SYN, SYN|ACK, ACK.

Capturing an IPv6 TCP handshake (just to show that IPv6 addresses are printed properly) yields the following:

```
solaris# tcpio.d
CPU           LADDR:LPORT                RADDR:RPORT  BYTES FLAGS
8   fe80::214:4fff:feed:d41c:22    <-  fe80::214:4fff:fe3b:76c8:45528    0 (SYN)
8   fe80::214:4fff:feed:d41c:22    ->  fe80::214:4fff:fe3b:76c8:45528    0 (SYN|ACK)
8   fe80::214:4fff:feed:d41c:22    <-  fe80::214:4fff:fe3b:76c8:45528    0 (ACK)
8   fe80::214:4fff:feed:d41c:22    <-  fe80::214:4fff:fe3b:76c8:45528    0 (ACK)
8   fe80::214:4fff:feed:d41c:22    <-  fe80::214:4fff:fe3b:76c8:45528   20 (PUSH|ACK)
```

Unfortunately, the IPv6 addresses are so long that they cause the output to overflow a width of 80 characters.[18]

Port Closed. Here a remote host attempted to connect to TCP port 123, which was closed:

```
solaris# tcpio.d
CPU          LADDR:LPORT                  RADDR:RPORT  BYTES FLAGS
8     192.168.2.145:123    <-    192.168.1.109:50708      0 (SYN)
8     192.168.2.145:123    ->    192.168.1.109:50708      0 (RST|ACK)
```

The server returned a TCP reset (RST).

Loopback Traffic. The following shows `tcpio.d` tracing a loopback connection to port 22:

```
solaris# tcpio.d
CPU          LADDR:LPORT                  RADDR:RPORT  BYTES FLAGS
0         127.0.0.1:41736  ->        127.0.0.1:22         0 (SYN)
0         127.0.0.1:22     <-        127.0.0.1:41736      0 (SYN)
0         127.0.0.1:22     ->        127.0.0.1:41736      0 (SYN|ACK)
0         127.0.0.1:41736  <-        127.0.0.1:22         0 (SYN|ACK)
0         127.0.0.1:41736  ->        127.0.0.1:22         0 (ACK)
0         127.0.0.1:22     <-        127.0.0.1:41736      0 (ACK)
0         127.0.0.1:22     ->        127.0.0.1:41736     20 (ACK)
0         127.0.0.1:41736  <-        127.0.0.1:22        20 (ACK)
0         127.0.0.1:41736  ->        127.0.0.1:22        20 (ACK)
0         127.0.0.1:22     <-        127.0.0.1:41736     20 (ACK)
0         127.0.0.1:41736  ->        127.0.0.1:22       504 (ACK)
0         127.0.0.1:22     <-        127.0.0.1:41736    504 (ACK)
[...]
0         127.0.0.1:41736  ->        127.0.0.1:22        32 (ACK)
0         127.0.0.1:22     <-        127.0.0.1:41736     32 (ACK)
0         127.0.0.1:41736  ->        127.0.0.1:22         0 (FIN|ACK)
0         127.0.0.1:22     <-        127.0.0.1:41736      0 (FIN|ACK)
0         127.0.0.1:22     ->        127.0.0.1:41736      0 (ACK)
0         127.0.0.1:41736  <-        127.0.0.1:22         0 (ACK)
^C
```

Since it is tracing at the TCP layer, it doesn't matter whether this TCP traffic is sent over a physical network interface: Everything can be observed.[19]

18. Staying within 80 characters is a strict tradition among Solaris kernel engineers.

19. Development versions of the tcp provider used different probes for TCP fusion, which is a Solaris performance feature that bypasses the TCP/IP stack for data packets on established TCP sessions. The final version of the provider rolled these into `tcp:::send` and `tcp:::receive`, since the provider interface should not expose Solaris implementation details to end users.

tcpbytes.d

This script shows which remote clients and local ports are performing how much I/O, in terms of TCP payload bytes.

Script

TCP payload bytes are calculated by taking the IP payload bytes and subtracting the TCP header. The size of the TCP header is available as `args[4]->tcp_offset`, which is the offset (in bytes) of the packet where TCP payload data begins.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing TCP payload bytes... Hit Ctrl-C to end.\n");
}

tcp:::receive
{
        @bytes[args[2]->ip_saddr, args[4]->tcp_dport] =
            sum(args[2]->ip_plength - args[4]->tcp_offset);
}

tcp:::send
{
        @bytes[args[2]->ip_daddr, args[4]->tcp_sport] =
            sum(args[2]->ip_plength - args[4]->tcp_offset);
}

dtrace:::END
{
        printf("  %-32s %-6s %16s\n", "REMOTE", "LPORT", "BYTES");
        printa("  %-32s %-6d %@16d\n", @bytes);
}
```

Script tcpbytes.d

Example

Here port 2049 (NFS) was the busiest, transferring about 40MB over TCP while this script was tracing.

```
# tcpbytes.d
Tracing TCP payload bytes... Hit Ctrl-C to end.
^C
  REMOTE                           LPORT             BYTES
  fe80::214:4fff:fe3b:76c8         23                  111
  192.168.2.8                      2049                164
  192.168.1.109                    22                  192
  192.168.100.4                    3260                384
  192.168.100.5                    3260                384
  192.168.2.53                     3260                768
```

```
    192.168.2.55                    3260              768
    192.168.2.156                   1001              840
    fe80::214:4fff:fe3b:76c8        22               5000
    192.168.1.109                   215             20727
    192.168.2.53                    2049         44048464
```

tcpsize.d

The `tcpsize.d script` shows the size of TCP sends and receives by client address and port. This could be used to identify whether a client was transferring data using many small I/Os or fewer larger I/Os.

Script

All tcp sends and receives are included in the output, including those for TCP packets that did not transfer data (ACKs, for example):

```
#!/usr/sbin/dtrace -s

tcp:::receive
{
        @bytes[args[2]->ip_saddr, args[4]->tcp_dport] =
            quantize(args[2]->ip_plength - args[4]->tcp_offset);
}

tcp:::send
{
        @bytes[args[2]->ip_daddr, args[4]->tcp_sport] =
            quantize(args[2]->ip_plength - args[4]->tcp_offset);
}
```

Script tcpsize.d

Example

The output has captured a couple of NFS clients performing I/O. The 192.168.100.4 client is performing TCP send/receives with sizes as large as 4KB to 8KB, whereas the 192.168.2.53 client reaches only between 1KB and 2KB. The difference here is known; the 192.168.100.4 client is using jumbo frames, whereas the other client is not. (Another reason for larger packets seen at the TCP level can be TCP large send offload, where TCP sends a large packet for the network card to fragment.) The counts seen for 0 bytes is an indication of how many TCP nonpayload packets were used.

```
server# tcpsize.d
dtrace: script './tcpsize.d' matched 8 probes
^C
[...]
```

continues

```
  192.168.100.4                                           2049
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |                                         13
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         8
             128 |@@@@@@@@@@@@@@@@@@@@                     3230
             256 |                                         0
             512 |                                         0
            1024 |                                         0
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@                     3220
            8192 |                                         0

  192.168.2.53                                            2049
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@                                  14903
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         4
             128 |@@@@@@@@@@@@@                            29778
             256 |                                         0
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@                     44661
            2048 |                                         0
```

tcpnmap.d

This is an example of examining event data to produce more information than just event counts. The `tcpnmap.d` script examines TCP events and flags to detect possible port scans from nmap[20] or similar port scanners.

The nmap port scanner is a powerful security tool for the analysis of host vulnerabilities. By varying TCP flags, you can perform various network port scans, including Xmas and null scans. The `tcpnmap.d` tool examines these flags to find traffic that may be scan events. However, they may also be normal traffic (as with the `connect()` scan). The differentiator is the volume of these suspicious events.

Script

This is a simple script, identifying different packet and event types and then populating a count aggregation with a descriptive string as the key.

20. *http://nmap.org*

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing for possible nmap scans... Hit Ctrl-C to end.\n");
}

tcp:::accept-refused
{
        @num["TCP_connect()_scan", args[2]->ip_daddr] = count();
}

tcp:::receive
/args[4]->tcp_flags == 0/
{
        @num["TCP_null_scan", args[2]->ip_saddr] = count();
}

tcp:::receive
/args[4]->tcp_flags == (TH_URG|TH_PUSH|TH_FIN)/
{
        @num["TCP_Xmas_scan", args[2]->ip_saddr] = count();
}

dtrace:::END
{
        printf("Possible scan events:\n\n");
        printf("   %-24s %-28s %8s\n", "TYPE", "HOST", "COUNT");
        printa("   %-24s %-28s %@8d\n", @num);
}
```

Script tcpnmap.d

Example

The `tcpnmap.d` script was run for ten seconds:

```
solaris# tcpnmap.d
Tracing for possible nmap scans... Hit Ctrl-C to end.
^C
Possible scan events:

   TYPE                     HOST                            COUNT
   TCP_null_scan            192.168.1.109                     208
   TCP_Xmas_scan            192.168.1.109                     304
   TCP_connect()_scan       192.168.1.109                     388
```

Here all our scan types had counts of more than 100, which (for this 10-second sample) is evidence of scanning.

tcpconnlat.d

This script measures TCP outbound connection latency. This is the time from the outbound SYN to the returned SYN|ACK and is a measure of network latency and remote host TCP processing time (time for the remote kernel to create the new TCP session and reply to the SYN).

Script

This script associates the `tcp:::connect-request` probe to the `tcp:::connect-established` probe through `args[1]->cs_cid`, which is a unique identifier for the connection.

```
#!/usr/sbin/dtrace -s

tcp:::connect-request
{
        start[args[1]->cs_cid] = timestamp;
}

tcp:::connect-established
/start[args[1]->cs_cid]/
{
        @latency["Connect Latency (ns)", args[2]->ip_daddr] =
            quantize(timestamp - start[args[1]->cs_cid]);
        start[args[1]->cs_cid] = 0;
}
```

Script tcpconnlat.d

Example

While the `tcpconnlat.d` script was running, several outbound TCP connections were performed.

```
solaris# tcpconnlat.d
dtrace: script './tcpconnlat.d' matched 2 probes
^C

  Connect Latency (ns)                              192.168.1.109
           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  62
          262144 |@                                        1
          524288 |                                         0

  Connect Latency (ns)                              72.5.124.61
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
        16777216 |                                         0
```

Connections to the nearby host 192.168.1.109 mostly completed with times between 0.13 ms and 0.26 ms. Connections to the Internet host 72.5.124.61 took longer, between 8 ms and 16 ms.

This DTrace script can be modified to provide the data in different ways, such as averages, or to print details of every connection as it occurs.

tcp1stbyte.d

This script is similar to `tcpconnlat.d` but measures TCP first-byte latency, which is the time from when the connection is established to when the first application data bytes arrive. This is a measure of both network latency and remote application load.

Script

This script is written in terms of the client initiating the connection, by beginning with the `tcp:::connect-established` probe. It could be modified for use on the server accepting the connection by changing the probe to `tcp:::accept-established`.

```
#!/usr/sbin/dtrace -s

tcp:::connect-established
{
        start[args[1]->cs_cid] = timestamp;
}

tcp:::receive
/start[args[1]->cs_cid] && (args[2]->ip_plength - args[4]->tcp_offset) > 0/
{
        @latency["1st Byte Latency (ns)", args[2]->ip_saddr] =
            quantize(timestamp - start[args[1]->cs_cid]);
        start[args[1]->cs_cid] = 0;
}
```

Script tcp1stbyte.d

The first-byte event is identified as the first `tcp:::receive` containing TCP payload bytes.

Example

Here connections to the same two remote hosts were performed as with `tcpconnlat.d` but with different results:

```
solaris# tcp1stbyte.d
dtrace: script 'tcp1stbyte.d' matched 6 probes
^C

  1st Byte Latency (ns)                              72.5.124.61
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
        33554432 |                                         0
```

continues

```
 1st Byte Latency (ns)                              192.168.1.109
           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |@@@@@@@@@@@@@@@@@@@                      12
          524288 |@@@@@@@@@@                               6
         1048576 |@@@@@                                    3
         2097152 |                                         0
         4194304 |@@                                       1
         8388608 |@@                                       1
        16777216 |                                         0
        33554432 |@@@                                      2
        67108864 |                                         0
```

The Internet host consistently takes 16 ms to 32 ms to return the first application data. The nearby host (192.168.1.109) often returns data faster than half a millisecond, but on a couple of occasions it took longer than 32 milliseconds.

The reasons why a target application sometimes returns slowly could be investigated using DTrace on the remote host.

tcp_rwndclosed.d

This script measures the time spent after the TCP receive window is advertised as zero. This stops the remote host from sending data, so high latency or low throughput suffered by this connection may be our own fault. The cause can be investigated further with DTrace; this script identifies whether zero-size received windows are being advertised and the time spent after a zero-size advertisement to when new data was received. This script and example were written by Alan Maguire, who has been developing other interesting and advanced scripts based on the tcp provider.[21]

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

tcp:::send
/ args[4]->tcp_window == 0 && (args[4]->tcp_flags & TH_RST) == 0 /
{
      rwndclosed[args[1]->cs_cid] = timestamp;
      rwndrnxt[args[1]->cs_cid] = args[3]->tcps_rnxt;
      @numrwndclosed[args[2]->ip_daddr, args[4]->tcp_dport] = count();
}
```

21. This script is currently at *http://blogs.sun.com/amaguire/entry/dtrace_tcp_provider_and_tcp* along with its companion for the send side; also see his blog, currently at *http://blogs.sun.com/amaguire*.

```
tcp:::receive
/ rwndclosed[args[1]->cs_cid] && args[4]->tcp_seq >= rwndrnxt[args[1]->cs_cid] /
{
        @meantimeclosed[args[2]->ip_saddr, args[4]->tcp_sport] =
            avg(timestamp - rwndclosed[args[1]->cs_cid]);
        @stddevtimeclosed[args[2]->ip_saddr, args[4]->tcp_sport] =
            stddev(timestamp - rwndclosed[args[1]->cs_cid]);
        rwndclosed[args[1]->cs_cid] = 0;
        rwndrnxt[args[1]->cs_cid] = 0;
}

END
{
        printf("%-20s %-8s %-25s %-8s %-8s\n",
            "Remote host", "Port", "TCP Avg RwndClosed(ns)", "StdDev",
            "Num");
        printa("%-20s %-8d %@-25d %@-8d %@-8d\n", @meantimeclosed,
            @stddevtimeclosed, @numrwndclosed);
}
```

Script tcp_rwndclosed.d

Example

Here, a high-resolution YouTube video was loaded in a browser:

```
solaris# dtrace -s tcp_rwndclosed.d
^C
Remote host          Port     TCP Avg RwndClosed(ns)    StdDev   Num
92.122.127.159       80       26914620                  0        1
```

This caused the receive window size to be advertised as zero and then the remote host to wait for 0.269 seconds before sending new data.

tcpfbtwatch.d

Monitoring inbound TCP connections can be useful for identifying how a server is being used and was achieved earlier with `tcpaccept.d` and `tcpacceptx.d`. The `tcpfbtwatch.d` script traces TCP accepts live and is an example of doing so via the fbt provider, should the tcp provider not be available.

This was written for a recent version of Solaris Nevada and is provided as an example of fbt tracing; it is not expected to run on other Solaris kernel versions.

Script (tcp Provider)

For comparison, this is how the script looks if the tcp provider is available:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz
```

continues

```

dtrace:::BEGIN
{
        printf("%-20s  %-24s %-24s %6s\n", "TIME", "REMOTE", "LOCAL", "LPORT");
}

tcp:::accept-established
{
        printf("%-20Y  %-24s %-24s %6d\n", walltimestamp,
            args[2]->ip_saddr, args[2]->ip_daddr, args[4]->tcp_dport);
}
```

Script tcpwatch.d (tcp provider)

Script (fbt Provider)

With the fbt provider, our DTracing job becomes much more difficult. On the plus side, there is only one place in the kernel code that completes accepting a TCP connection: It happens inside `tcp_rput_data()`. However, another function needs to be traced to dig out the IPv6 details correctly: `tcp_find_pktinfo()`.

The `inet_ntoa()`, `inet_ntoa6()`, and `ntohs()` functions were used; if they, too, are unavailable, see how they can be performed manually, as in `ipfbtsnoop.d`:

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option switchrate=10hz

#define IPH_HDR_VERSION(ipha) \
        ((int)(((ipha_t *)ipha)->ipha_version_and_hdr_length) >> 4)

#define TCPS_SYN_RCVD   -1

#define conn_tcp        conn_proto_priv.cp_tcp
#define conn_lport      u_port.tcpu_ports.tcpu_lport

dtrace:::BEGIN
{
        printf("%-20s  %-24s %-24s %6s\n", "TIME", "REMOTE", "LOCAL", "LPORT");
}

fbt::tcp_rput_data:entry
{
        self->connp = (conn_t *)arg0;
        self->tcp = self->connp->conn_tcp;
        self->mp = args[1];
        self->ipha = (ipha_t *)self->mp->b_rptr;
        self->in_tcp_rput_data = 1;
}

fbt::tcp_rput_data:entry
/self->tcp->tcp_state == TCPS_SYN_RCVD && IPH_HDR_VERSION(self->ipha) == 4/
{
        this->src = inet_ntoa(&self->ipha->ipha_src);
        this->dst = inet_ntoa(&self->ipha->ipha_dst);
        this->lport = ntohs(self->connp->conn_lport);
        printf("%-20Y  %-24s %-24s %6d\n", walltimestamp, this->src,
            this->dst, this->lport);
}
```

```
37
38  fbt::tcp_find_pktinfo:return
39  /self->in_tcp_rput_data && self->tcp->tcp_state == TCPS_SYN_RCVD &&
40      IPH_HDR_VERSION(self->ipha) == 6/
41  {
42          this->mp = args[1];
43          this->ip6h = (struct ip6_hdr *)this->mp->b_rptr;
44          this->src = inet_ntoa6(&this->ip6h->ip6_src);
45          this->dst = inet_ntoa6(&this->ip6h->ip6_dst);
46          this->lport = ntohs(self->connp->conn_lport);
47          printf("%-20Y  %-24s %-24s %6d\n", walltimestamp, this->src,
48              this->dst, this->lport);
49  }
50
51  fbt::tcp_rput_data:return
52  {
53          self->connp = 0; self->tcp = 0; self->mp = 0;
54          self->ipha = 0; self->in_tcp_rput_data = 0;
55  }
```

Script tcpfbtwatch.d

Lines 10 and 11 are from the header definition for `conn_t` in the source code (`uts/common/inet/ipclassifier.h`) and are how the `tcp_t` and local port information are fetched when starting from a `conn_t`. The `tcp_t` is used to retrieve the current TCP session state, as tested on lines 29 and 39.

When the `tcp_rput_data()` or `tcp_find_pktinfo()` functions change in the kernel code, it's likely that this script will need adjustments to match the changes to continue working.

Example

```
solaris# tcpfbtwatch.d
TIME                  REMOTE                    LOCAL                      LPORT
2010 Jan 17 07:44:50  192.168.1.109             192.168.2.145                 22
2010 Jan 17 07:44:51  fe80::214:4fff:fe3b:76c8 fe80::214:4fff:feed:d41c     22
2010 Jan 17 07:44:55  192.168.1.109             192.168.2.145                 80
2010 Jan 17 07:44:59  192.168.1.109             192.168.2.145                 22
2010 Jan 17 07:45:02  192.168.1.109             192.168.2.145                215
2010 Jan 17 07:45:08  192.168.1.188             192.168.2.145                 22
^C
```

While tracing, several inbound TCP connections were established, mostly to port 22 (`ssh`). One connection was using the IPv6 protocol.

tcpsnoop.d

The `tcpsnoop.d` script traces TCP sends and receives with process details. It was written to produce output similar to the Solaris `snoop(1M)` utility (`tcpdump(1)` on other operating systems), which traces packets on a given interface. `tcpsnoop.d` includes details of the processes responsible for sending or receiving those packets.

When tcpsnoop.d was first written, this was difficult for a number of reasons:[22]

- No stable network providers existed; only fbt was available.
- Packets are not received in process context (see the "Common Mistakes" section).
- Packets are often not sent in process context because of buffering.
- The Solaris TCP/IP stack often uses advanced programming features such as function pointers, which can make reading and understanding code more difficult.
- The Solaris TCP/IP stack is a large body of code to wade through: more than 100,000 lines.
- Network packets can be processed by many different code paths in TCP/IP.
- The Solaris TCP/IP stack implementation changes regularly with kernel updates.

Since the fbt provider can trace all of the kernel functions, it should still be possible to write tcpsnoop despite these difficulties.

The final result is best shown with the following example:

```
solaris# tcpsnoop.d
  UID    PID LADDR           LPORT DR RADDR           RPORT  SIZE CMD
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 <- 192.168.1.1        79    66 finger
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    56 finger
  100  20892 192.168.1.5     36398 <- 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 <- 192.168.1.1        79   606 finger
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 <- 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 -> 192.168.1.1        79    54 finger
  100  20892 192.168.1.5     36398 <- 192.168.1.1        79    54 finger
    0    242 192.168.1.5        23 <- 192.168.1.1     54224    54 inetd
    0    242 192.168.1.5        23 -> 192.168.1.1     54224    54 inetd
    0    242 192.168.1.5        23 <- 192.168.1.1     54224    54 inetd
    0    242 192.168.1.5        23 <- 192.168.1.1     54224    78 inetd
    0    242 192.168.1.5        23 -> 192.168.1.1     54224    54 inetd
    0  20893 192.168.1.5        23 -> 192.168.1.1     54224    57 in.telnetd
    0  20893 192.168.1.5        23 <- 192.168.1.1     54224    54 in.telnetd
    0  20893 192.168.1.5        23 -> 192.168.1.1     54224    78 in.telnetd
    0  20893 192.168.1.5        23 <- 192.168.1.1     54224    57 in.telnetd
    0  20893 192.168.1.5        23 -> 192.168.1.1     54224    54 in.telnetd
    0  20893 192.168.1.5        23 <- 192.168.1.1     54224    54 in.telnetd
```

22. And there was an eighth reason: tcpsnoop.d was first written *without access to the source code* because OpenSolaris was not yet public.

```
     0  20893 192.168.1.5        23 -> 192.168.1.1     54224    60 in.telnetd
     0  20893 192.168.1.5        23 <- 192.168.1.1     54224    63 in.telnetd
     0  20893 192.168.1.5        23 -> 192.168.1.1     54224    54 in.telnetd
     0  20893 192.168.1.5        23 <- 192.168.1.1     54224    60 in.telnetd
     0  20893 192.168.1.5        23 -> 192.168.1.1     54224    60 in.telnetd
     0  20893 192.168.1.5        23 <- 192.168.1.1     54224    60 in.telnetd
     0  20893 192.168.1.5        23 -> 192.168.1.1     54224    72 in.telnetd
[...]
```

We matched the correct process for the outbound finger command, and for the inbound telnet connection, even as the socket file descriptor is passed from `inetd` to `in.telnetd`.

Soon after `tcpsnoop.d` was written, a kernel update changed some of the underlying functions it was tracing, and it stopped working. It was fixed but stopped working again after another kernel update. So far, it has been broken several times because of kernel updates.[23] We have pointed out the dangers of using the unstable fbt provider throughout this book; `tcpsnoop.d` is a prime example of those dangers.

`tcpsnoop.d` has now been rewritten using the stable tcp provider and is shown at the end of this script section. Until your operating system has the tcp provider, treat the fbt-based `tcpsnoop.d` not as a script you can use as is but as a project with sample solutions provided. The DTraceToolkit contains two versions: `tcpsnoop.d` (which is currently still the fbt-based version) for some early versions of Solaris 10, and `tcpsnoop_snv.d` (also fbt based) for recent versions of Solaris Nevada (snv). There are plenty of Solaris 10 and OpenSolaris versions for which neither fbt-based `tcpsnoop.d` version will work.

Here we will explain the `tcpsnoop_snv.d` script line by line, as an example of advanced DTrace. Be warned: This is the longest, most difficult, and most brittle script I've ever written; if any of those attributes are unacceptable, wait until the tcp provider is available on your operating system, and use the tcp provider–based version of this script instead.

Script: fbt Based

This is the full script for `tcpsnoop_snv.d`, with some comments from the header truncated to save space. Because of its length and complexity, the script is presented in multiple sections. See the DTraceToolkit for the full version (and other versions):

23. Many thanks to the DTrace community on *dtrace-discuss@opensolaris.org* who have posted updates to `tcpsnoop.d` to keep it working on various kernel versions.

```
1  #!/usr/sbin/dtrace -Cs
[...]
19   * $Id: tcpsnoop_snv.d 69 2007-10-04 13:40:00Z brendan $
[...]
64   */
65
66  #pragma D option quiet
67  #pragma D option switchrate=10hz
68
69  #include <sys/file.h>
70  #include <inet/common.h>
71  #include <sys/byteorder.h>
72
73  /*
74   * Print header
75   */
76  dtrace:::BEGIN
77  {
78          /* print main headers */
79          printf("%5s %6s %-15s %5s %2s %-15s %5s %5s %s\n",
80              "UID", "PID", "LADDR", "LPORT", "DR", "RADDR", "RPORT",
81              "SIZE", "CMD");
82  }
83
```

Script tcpsnoop_snv.d

The goal of lines 84 to 110 is to store process information (`execname`, `pid`, `uid`) that can be retrieved during TCP function calls. A TCP connection event begins as a network interrupt, at which point the accepting process is not on-CPU (it is sleeping). Once the packet has been processed, the kernel switches to the target process thread that was accepting the connection. At that point, the process information (`execname`, `pid`, `uid`) for the connection is valid and can be stored for later lookup. To wait for this to occur, tracing is performed in the socket layer, which is assumed to be after the context switch back to the accepting thread:

```
84   /*
85    * TCP Process inbound connections
86    *
87    * 0x00200000 has been hardcoded. It was SS_TCP_FAST_ACCEPT, but was
88    * renamed to SS_DIRECT around build 31.
89    */
90   fbt:sockfs:sotpi_accept:entry
91   /(arg1 & FREAD) && (arg1 & FWRITE) && (args[0]->so_state & 0x00200000)/
92   {
93           self->sop = args[0];
94   }
95
96   fbt:sockfs:sotpi_create:return
97   /self->sop/
98   {
99           self->nsop = (struct sonode *)arg1;
100  }
101
102  fbt:sockfs:sotpi_accept:return
103  /self->nsop/
```

```
104  {
105          this->tcpp = (tcp_t *)self->nsop->so_priv;
106          self->connp = (conn_t *)this->tcpp->tcp_connp;
107          tname[(int)self->connp] = execname;
108          tpid[(int)self->connp] = pid;
109          tuid[(int)self->connp] = uid;
110  }
```

Script tcpsnoop_snv.d (continued)

The socket accept function is traced: `sotpi_accept()`. During `sotpi_accept()`, a call to `sotpi_create()` will return a `struct sonode` for the socket, which contains useful data. We can retrieve it by tracing `sotpi_create:return` and saving the return value, `arg1`.

The useful data of `struct sonode` can be seen on lines 105 and 106, where a `conn_t` pointer is retrieved from the socket node. The `conn_t` pointer is used as a unique ID; specifically, the memory address of the `conn_t` is used as a unique ID. No other `conn_t`s will refer (or can refer) to the same memory address at the same time. `conn_t` is available in TCP functions, so it functions as a unique ID that can bridge socket and TCP events.

This ID is used as a key in three associative arrays: `tname`, `tpid`, and `tuid`. These translate the `conn_t` pointer address to the process `execname`, `pid`, and `uid`, which will be retrieved from TCP later.

```
111
112  fbt:sockfs:sotpi_accept:return
113  {
114          self->nsop = 0;
115          self->sop = 0;
116  }
117
118  /*
119   * TCP Process outbound connections
120   */
121  fbt:ip:tcp_connect:entry
122  {
123          this->tcpp = (tcp_t *)arg0;
124          self->connp = (conn_t *)this->tcpp->tcp_connp;
125          tname[(int)self->connp] = execname;
126          tpid[(int)self->connp] = pid;
127          tuid[(int)self->connp] = uid;
128  }
```

Script tcpsnoop_snv.d (continued)

This stores the same associative arrays as before but for outbound TCP events. This is a different code path and scenario and is approached differently. Here, we assume that the correct process is still on-CPU by the time we reach `tcp_connect()`. At that point, the process information can be cached with the available `conn_t` pointer. Processing socket events wasn't needed (although if `tcp_connect()`

changes in an update and can then occur outside process context, process information will need to be passed from socket to TCP as for TCP inbound connections).

```
129
130 /*
131  * TCP Data translations
132  */
133 fbt:sockfs:sotpi_accept:return,
134 fbt:ip:tcp_connect:return
135 /self->connp/
136 {
137         /* fetch ports */
138 #if defined(_BIG_ENDIAN)
139         self->lport = self->connp->u_port.tcpu_ports.tcpu_lport;
140         self->fport = self->connp->u_port.tcpu_ports.tcpu_fport;
141 #else
142         self->lport = BSWAP_16(self->connp->u_port.tcpu_ports.tcpu_lport);
143         self->fport = BSWAP_16(self->connp->u_port.tcpu_ports.tcpu_fport);
144 #endif
```

Script tcpsnoop_snv.d (continued)

Lines 138 to 144 convert ports from network byte order to host byte order. This was necessary for the script to work on both Solaris x86 and Solaris SPARC. We can now use DTrace's `ntohs()` function to do this.

```
145
146         /* fetch IPv4 addresses */
147         this->fad12 =
148             (int)self->connp->connua_v6addr.connua_faddr._S6_un._S6_u8[12];
149         this->fad13 =
150             (int)self->connp->connua_v6addr.connua_faddr._S6_un._S6_u8[13];
151         this->fad14 =
152             (int)self->connp->connua_v6addr.connua_faddr._S6_un._S6_u8[14];
153         this->fad15 =
154             (int)self->connp->connua_v6addr.connua_faddr._S6_un._S6_u8[15];
155         this->lad12 =
156             (int)self->connp->connua_v6addr.connua_laddr._S6_un._S6_u8[12];
157         this->lad13 =
158             (int)self->connp->connua_v6addr.connua_laddr._S6_un._S6_u8[13];
159         this->lad14 =
160             (int)self->connp->connua_v6addr.connua_laddr._S6_un._S6_u8[14];
161         this->lad15 =
162             (int)self->connp->connua_v6addr.connua_laddr._S6_un._S6_u8[15];
163
164         /* convert type for use with lltostr() */
165         this->fad12 = this->fad12 < 0 ? 256 + this->fad12 : this->fad12;
166         this->fad13 = this->fad13 < 0 ? 256 + this->fad13 : this->fad13;
167         this->fad14 = this->fad14 < 0 ? 256 + this->fad14 : this->fad14;
168         this->fad15 = this->fad15 < 0 ? 256 + this->fad15 : this->fad15;
169         this->lad12 = this->lad12 < 0 ? 256 + this->lad12 : this->lad12;
170         this->lad13 = this->lad13 < 0 ? 256 + this->lad13 : this->lad13;
171         this->lad14 = this->lad14 < 0 ? 256 + this->lad14 : this->lad14;
172         this->lad15 = this->lad15 < 0 ? 256 + this->lad15 : this->lad15;
173
174         /* stringify addresses */
175         self->faddr = strjoin(lltostr(this->fad12), ".");
176         self->faddr = strjoin(self->faddr, strjoin(lltostr(this->fad13), "."));
```

```
177         self->faddr = strjoin(self->faddr, strjoin(lltostr(this->fad14), "."));
178         self->faddr = strjoin(self->faddr, lltostr(this->fad15 + 0));
179         self->laddr = strjoin(lltostr(this->lad12), ".");
180         self->laddr = strjoin(self->laddr, strjoin(lltostr(this->lad13), "."));
181         self->laddr = strjoin(self->laddr, strjoin(lltostr(this->lad14), "."));
182         self->laddr = strjoin(self->laddr, lltostr(this->lad15 + 0));
```

Script tcpsnoop_snv.d (continued)

Lines 147 to 182 retrieve and stringify IPv4 addresses. This script was written before the `inet_ntoa()` function existed for DTrace, and so these conversions are performed manually.

```
183
184         /* fix direction and save values */
185         tladdr[(int)self->connp] = self->laddr;
186         tfaddr[(int)self->connp] = self->faddr;
187         tlport[(int)self->connp] = self->lport;
188         tfport[(int)self->connp] = self->fport;
```

Script tcpsnoop_snv.d (continued)

The stringified addresses and port numbers are saved in more associative arrays for lookup later. The `self->` (thread-local) variables aren't used outside this clause and really should be `this->` (clause-local) variables. However, the initial version of DTrace didn't allow clause-local variables to contain strings, so thread-local variables were used instead.

```
189
190         /* all systems go */
191         tok[(int)self->connp] = 1;
```

Script tcpsnoop_snv.d (continued)

Remember that we cached the addresses and ports for later lookup.

```
192  }
193
194  /*
195   * TCP Clear connp
196   */
197  fbt:ip:tcp_get_conn:return
198  {
199         /* Q_TO_CONN */
200         this->connp = (conn_t *)arg1;
201         tok[(int)this->connp] = 0;
202         tpid[(int)this->connp] = 0;
```

continues

```
        tuid[(int)this->connp] = 0;
        tname[(int)this->connp] = 0;
}
```

Script tcpsnoop_snv.d (continued)

conn_ts can be reused, which would leave stale data in the associative arrays that are keyed by their addresses. To prevent this, the associative arrays are cleared whenever conn_ts are first retrieved.

```

/*
 * TCP Process "port closed"
 */
fbt:ip:tcp_xmit_early_reset:entry
{
        this->queuep = args[7]->tcps_g_q;
        this->connp = (conn_t *)this->queuep->q_ptr;
        this->tcpp = (tcp_t *)this->connp->conn_tcp;

        /* split addresses */
        this->ipha = (ipha_t *)args[1]->b_rptr;
        this->fad15 = (this->ipha->ipha_src & 0xff000000) >> 24;
        this->fad14 = (this->ipha->ipha_src & 0x00ff0000) >> 16;
        this->fad13 = (this->ipha->ipha_src & 0x0000ff00) >> 8;
        this->fad12 = (this->ipha->ipha_src & 0x000000ff);
        this->lad15 = (this->ipha->ipha_dst & 0xff000000) >> 24;
        this->lad14 = (this->ipha->ipha_dst & 0x00ff0000) >> 16;
        this->lad13 = (this->ipha->ipha_dst & 0x0000ff00) >> 8;
        this->lad12 = (this->ipha->ipha_dst & 0x000000ff);

        /* stringify addresses */
        self->faddr = strjoin(lltostr(this->fad12), ".");
        self->faddr = strjoin(self->faddr, strjoin(lltostr(this->fad13), "."));
        self->faddr = strjoin(self->faddr, strjoin(lltostr(this->fad14), "."));
        self->faddr = strjoin(self->faddr, lltostr(this->fad15 + 0));
        self->laddr = strjoin(lltostr(this->lad12), ".");
        self->laddr = strjoin(self->laddr, strjoin(lltostr(this->lad13), "."));
        self->laddr = strjoin(self->laddr, strjoin(lltostr(this->lad14), "."));
        self->laddr = strjoin(self->laddr, lltostr(this->lad15 + 0));

        self->reset = 1;
}

/*
 * TCP Fetch "port closed" ports
 */
fbt:ip:tcp_xchg:entry
/self->reset/
{
#if defined(_BIG_ENDIAN)
        self->lport = (uint16_t)arg0;
        self->fport = (uint16_t)arg1;
#else
        self->lport = BSWAP_16((uint16_t)arg0);
        self->fport = BSWAP_16((uint16_t)arg1);
#endif
        self->lport = BE16_TO_U16(arg0);
        self->fport = BE16_TO_U16(arg1);
}
```

```
256
257 /*
258  * TCP Print "port closed"
259  */
260 fbt:ip:tcp_xmit_early_reset:return
261 {
262         self->name = "<closed>";
263         self->pid = 0;
264         self->uid = 0;
265         self->size = 54;        /* should check trailers */
266         self->dir = "<-";
267         printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
268             self->uid, self->pid, self->laddr, self->lport, self->dir,
269             self->faddr, self->fport, self->size, self->name);
270         self->dir = "->";
271         printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
272             self->uid, self->pid, self->laddr, self->lport, self->dir,
273             self->faddr, self->fport, self->size, self->name);
274         self->reset = 0;
275         self->size = 0;
276         self->name = 0;
277 }
```

Script tcpsnoop_snv.d (continued)

Lines 210 to 277 process inbound connections to closed ports (TCP returns RST). This code exists only so that `tcpsnoop.d` can see attempted connections to closed ports and print the inbound request and the outbound reset. These lines have broken in the past because of kernel updates where the `tcp_xchg()` function was changed. A simple workaround was to delete these lines from `tcpsnoop.d`, if you don't care about seeing TCP RSTs. Another problem was only spotted during review of this chapter; lines 253 and 254 overwrite the previous (correct) port values and should be dropped. (This may be evidence that long and complex D scripts aren't just difficult to maintain; they are difficult to get right in the first place. This would have been easier to spot in a much shorter script.)

```

/*
 * TCP Process Write
 */
fbt:ip:tcp_send_data:entry
{
        self->conn_p = (conn_t *)args[0]->tcp_connp;
}

fbt:ip:tcp_send_data:entry
/tok[(int)self->conn_p]/
{
        self->dir = "->";
        self->size = msgdsize(args[2]) + 14;    /* should check trailers */
        self->uid = tuid[(int)self->conn_p];
        self->laddr = tladdr[(int)self->conn_p];
        self->faddr = tfaddr[(int)self->conn_p];
        self->lport = tlport[(int)self->conn_p];
        self->fport = tfport[(int)self->conn_p];
```

continues

```
297           self->ok = 2;
298
299           /* follow inetd -> in.* transitions */
300           self->name = pid && (tname[(int)self->conn_p] == "inetd") ?
301               execname : tname[(int)self->conn_p];
302           self->pid = pid && (tname[(int)self->conn_p] == "inetd") ?
303               pid : tpid[(int)self->conn_p];
304           tname[(int)self->conn_p] = self->name;
305           tpid[(int)self->conn_p] = self->pid;
306   }
```

Script tcpsnoop_snv.d (continued)

Lines 282 to 306 process TCP sends. The `tcp_send_data()` function can retrieve the `conn_t` pointer from its first argument, the `args[0] tcp_t`. Using the `conn_t` pointer, the process information is retrieved from various associative arrays and saved as thread-local variables for later printing.

Some minor notes: The comment on line 291 mentions checking trailers; this should be mentioning padding, not trailers. And the `+ 14` adds the size of the Ethernet header, which may be better coded as `sizeof (struct ether_header)`, because it both returns 14 and makes it obvious in the D program what this is referring to. (However, the Ethernet header may be bigger with VLANs, which this does not check for.)

There's some special casing for `inetd` on lines 300 to 303 so that the process that `inetd` hands to the connection is followed.

```
307
308   /*
309    * TCP Process Read
310    */
311   fbt:ip:tcp_rput_data:entry
312   {
313           self->conn_p = (conn_t *)arg0;
314           self->size = msgdsize(args[1]) + 14;    /* should check trailers */
```

Script tcpsnoop_snv.d (continued)

The +14 (line 314) adds the assumed size of the Ethernet header again so that the output of `tcpsnoop.d` matches the output of `snoop -S`.

```
315   }
316
317   fbt:ip:tcp_rput_data:entry
318   /tok[(int)self->conn_p]/
319   {
320           self->dir = "<-";
321           self->uid = tuid[(int)self->conn_p];
322           self->laddr = tladdr[(int)self->conn_p];
323           self->faddr = tfaddr[(int)self->conn_p];
324           self->lport = tlport[(int)self->conn_p];
```

```
325         self->fport = tfport[(int)self->conn_p];
326         self->ok = 2;
327
328         /* follow inetd -> in.* transitions */
329         self->name = pid && (tname[(int)self->conn_p] == "inetd") ?
330             execname : tname[(int)self->conn_p];
331         self->pid = pid && (tname[(int)self->conn_p] == "inetd") ?
332             pid : tpid[(int)self->conn_p];
333         tname[(int)self->conn_p] = self->name;
334         tpid[(int)self->conn_p] = self->pid;
335 }
```

Script tcpsnoop_snv.d (continued)

For TCP receives, lines 311 to 335 retrieve the process and IP data for this connection and store it in thread-local variables ready to print. It can be retrieved since `tcp_rput_data()` has the `conn_t` as `arg0`, which is the key to various associative arrays containing that data.

Some special casing exists for `inetd` on lines 329 to 332 so that we can follow the process to which `inetd` hands the connection.

```
336
337 /*
338  * TCP Complete printing outbound handshake
339  */
340 fbt:ip:tcp_connect:return
341 /self->connp/
342 {
343         self->name = tname[(int)self->connp];
344         self->pid = tpid[(int)self->connp];
345         self->uid = tuid[(int)self->connp];
346         self->size = 54;        /* should check trailers */
347         self->dir = "->";
348         /* this packet occured before connp was fully established */
349         printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
350             self->uid, self->pid, self->laddr, self->lport, self->dir,
351             self->faddr, self->fport, self->size, self->name);
352 }
```

Script tcpsnoop_snv.d (continued)

Lines 340 to 352 are a special case for the final ACK in an outbound TCP connection. Since the packet is sent before the `conn_t` is fully initialized by the kernel, this packet will not be picked up by the usual tracing based on `conn_t`. It is printed here separately.

```
353
354 /*
355  * TCP Complete printing inbound handshake
356  */
357 fbt:sockfs:sotpi_accept:return
358 /self->connp/
```

continues

```
359  {
360          self->name = tname[(int)self->connp];
361          self->pid = tpid[(int)self->connp];
362          self->uid = tuid[(int)self->connp];
363          self->size = 54;        /* should check trailers */
364          /* these packets occured before connp was fully established */
365          self->dir = "<-";
366          printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
367              self->uid, self->pid, self->laddr, self->lport, self->dir,
368              self->faddr, self->fport, self->size, self->name);
369          self->dir = "->";
370          printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
371              self->uid, self->pid, self->laddr, self->lport, self->dir,
372              self->faddr, self->fport, self->size, self->name);
373          self->dir = "<-";
374          printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
375              self->uid, self->pid, self->laddr, self->lport, self->dir,
376              self->faddr, self->fport, self->size, self->name);
377  }
```

Script tcpsnoop_snv.d (continued)

Lines 357 to 377 traces the TCP handshake for inbound connections, which became a complex problem: The accepting process doesn't step on-CPU until the handshake is complete. Since that's the case, how do we print out earlier lines for the TCP packets if we don't yet have the process information cached from the socket layer?

The answer was to cheat: A complete three-way handshake is printed when the third packet is received, and `sotpi_accept()` returns. Since we know that this connection was established, we can guess that the earlier two packets were the SYN and SYN|ACK and print them with the now-available process information.

```
378
379  /*
380   * Print output
381   */
382  fbt:ip:tcp_send_data:entry,
383  fbt:ip:tcp_rput_data:entry
384  /self->ok == 2/
385  {
386          /* print output line */
387          printf("%5d %6d %-15s %5d %2s %-15s %5d %5d %s\n",
388              self->uid, self->pid, self->laddr, self->lport, self->dir,
389              self->faddr, self->fport, self->size, self->name);
390  }
```

Script tcpsnoop_snv.d (continued)

Lines 387 to 389 print a line of output for this TCP I/O. Most of the TCP send/receives are printed by this section of code.

Finally, we clean up variables used earlier.

```

/*
 * TCP Clear connect variables
 */
fbt:sockfs:sotpi_accept:return,
fbt:ip:tcp_connect:return
/self->connp/
{
        self->faddr = 0;
        self->laddr = 0;
        self->fport = 0;
        self->lport = 0;
        self->connp = 0;
        self->name = 0;
        self->pid = 0;
        self->uid = 0;
}

/*
 * TCP Clear r/w variables
 */
fbt:ip:tcp_send_data:entry,
fbt:ip:tcp_rput_data:entry
{
        self->ok = 0;
        self->dir = 0;
        self->uid = 0;
        self->pid = 0;
        self->size = 0;
        self->name = 0;
        self->lport = 0;
        self->fport = 0;
        self->laddr = 0;
        self->faddr = 0;
        self->conn_p = 0;
}
```

Script tcpsnoop_snv.d

As you can see, we've gone to a lot of effort to ensure that `tcpsnoop.d` traces every packet, even those that may not be interesting (TCP handshakes, RSTs) so that the output matches, line by line, the output of `snoop`. Was the effort worth it? Some of the extra code has made `tcpsnoop.d` more brittle during kernel updates. The key problem that `tcpsnoop.d` can solve is to identify processes responsible for network packets, which it can do without tracing TCP handshakes or TCP RSTs.

Script: tcp-Based

The following is `tcpsnoop.d`, written using the stable tcp provider. This version is shipped under `/usr/demo/dtrace` in Solaris Nevada (which will become OpenSolaris):

```
#!/usr/sbin/dtrace -s
/*
[...header truncated...]
 */
```

continues

```

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%6s %6s %15s:%-5s      %15s:%-5s %6s %s\n",
            "TIME", "PID", "LADDR", "PORT", "RADDR", "PORT", "BYTES", "FLAGS");
}

tcp:::send
{
        this->length = args[2]->ip_plength - args[4]->tcp_offset;
        printf("%6d %6d %15s:%-5d  ->  %15s:%-5d %6d (",
            timestamp/1000, args[1]->cs_pid, args[2]->ip_saddr,
            args[4]->tcp_sport, args[2]->ip_daddr, args[4]->tcp_dport,
            this->length);
}

tcp:::receive
{
        this->length = args[2]->ip_plength - args[4]->tcp_offset;
        printf("%6d %6d %15s:%-5d  <-  %15s:%-5d %6d (",
            timestamp/1000, args[1]->cs_pid, args[2]->ip_daddr,
            args[4]->tcp_dport, args[2]->ip_saddr, args[4]->tcp_sport,
            this->length);
}

tcp:::send,
tcp:::receive
{
        printf("%s", args[4]->tcp_flags & TH_FIN ? "FIN|" : "");
        printf("%s", args[4]->tcp_flags & TH_SYN ? "SYN|" : "");
        printf("%s", args[4]->tcp_flags & TH_RST ? "RST|" : "");
        printf("%s", args[4]->tcp_flags & TH_PUSH ? "PUSH|" : "");
        printf("%s", args[4]->tcp_flags & TH_ACK ? "ACK|" : "");
        printf("%s", args[4]->tcp_flags & TH_URG ? "URG|" : "");
        printf("%s", args[4]->tcp_flags & TH_ECE ? "ECE|" : "");
        printf("%s", args[4]->tcp_flags & TH_CWR ? "CWR|" : "");
        printf("%s", args[4]->tcp_flags == 0 ? "null " : "");
        printf("\b)\n");
}
```

Script tcpsnoop.d

Obviously this is a much shorter script and one that will continue to work with kernel updates. Note that this version does not trace the `inetd` process hand-off. In the six years that have passed since the original `tcpsnoop.d` was written, the `inetd` process has become less frequently used in production systems (for example, `sshd` instead of `in.telnetd`, `proftpd` instead of `in.ftpd`, and so on), and the extra complexity of tracing `inetd` may no longer be important. It can always be added again if needed.

Another difference is that the `execname` is lost from the output, which only contains the PID. There currently isn't a simple and stable way to fix this, such as a D function or array for converting a PID to the execname; if and when there is, this script can be updated. If this was desired in the meantime, other techniques can be used (such as caching pid to execname mappings in an associative array or digging it out of kernel structures).

UDP Scripts

The User Datagram Protocol (RFC 768) is a simple protocol for use by applications that do not require the reliability (and overhead) that TCP (or SCTP) provides. Scripts can be written to trace UDP using the udp provider (if available) and the mib and syscall providers for an idea of UDP usage across the system. To examine the internal operation of UDP in the network stack, the unstable fbt provider can be used.

udp provider

Listing udp provider probes (Solaris Nevada, circa June 2010) yields the following:

```
solaris# dtrace -ln udp:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
14125        udp                ip                          udp_send send
14126        udp                ip                 udp_output_newdst send
14127        udp                ip                          udp_wput send
14128        udp                ip                udp_output_lastdst send
14129        udp                ip              udp_output_connected send
14130        udp                ip              udp_output_ancillary send
14138        udp                ip                         udp_input receive
```

The provider is simple, like UDP itself: It has probes only for send and receive. An example of using these is given in the scripts that follow.

The udp provider is one of the newest (integrated into Solaris Nevada build 142) and may not be available in your operating system. If not, UDP events can be traced in the kernel using the mib provider (if available) and the fbt provider.

fbt Provider

fbt is an unstable interface: It exports kernel functions and data structures that may change from release to release. However, it does offer complete observability into the internals of UDP in the network stack and may be used if the udp provider is not available or does not provide the visibility desired.

Listing the fbt probes for udp functions on Solaris Nevada, circa December 2009 (we are deliberately switching to an older Solaris version, before the udp provider was available):

```
solaris# dtrace -ln 'fbt::udp_*:'
   ID   PROVIDER            MODULE                          FUNCTION NAME
56675        fbt                ip              udp_conn_constructor entry
56676        fbt                ip              udp_conn_constructor return
56677        fbt                ip               udp_conn_destructor entry
56678        fbt                ip               udp_conn_destructor return
57272        fbt                ip            udp_get_next_priv_port entry
```

continues

```
57273       fbt              ip            udp_get_next_priv_port return
57274       fbt              ip              udp_bind_hash_report entry
57275       fbt              ip              udp_bind_hash_report return
[...truncated...]
```

One hundred fifty-four probes were listed for tracing the internals of UDP on this kernel version.

To get an idea of the udp functions at play, we'll count probes that fire with a load of 10,000 1KB UDP writes:

```
solaris# dtrace -n 'fbt::udp_*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::udp_*:entry ' matched 77 probes
^C

  udp_bind                                                          1
  udp_bind_ack                                                      1
  udp_bind_hash_insert                                              1
  udp_bind_result                                                   1
  udp_update_next_port                                              1
  udp_capability_req                                                2
  udp_copy_info                                                     2
  udp_bind_hash_remove                                              4
  udp_close                                                         4
  udp_close_free                                                    4
  udp_conn_constructor                                              4
  udp_conn_destructor                                               4
  udp_open                                                          4
  udp_openv4                                                        4
  udp_quiesce_conn                                                  4
  udp_rcv_drain                                                     4
  udp_set_rcv_hiwat                                                 4
  udp_wput_iocdata                                                  9
  udp_wput_other                                                   16
  udp_output_v4                                                 10006
  udp_send_data                                                 10006
  udp_xmit                                                      10006
  udp_wput                                                      10022
```

The functions with higher counts (in the 10,000s) are likely to be those processing I/O and those with lower counts (less than 10) are those that initiate the session. Perform this experiment in the opposite direction (or trace on the remote host) to see the UDP receive side.

The relationship between these functions can be illustrated by examining stack traces, as shown in the "fbt Provider" section. Another way to learn the fbt probes is to map known mib events to the fbt functions, as demonstrated in the "mib Provider" section. And of course, if the source is available, it provides the best reference for the fbt probes and arguments.

udpstat.d

Various UDP statistics are traced from the mib provider and printed every second, similarly to the `ipstat.d` script:

Script

A variable called `line` is used to track when to reprint the header. This happens every 20 lines; without it, the screen could fill with numbers and become difficult to follow.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           LINES = 20; line = 0;
8   }
9
10  profile:::tick-1sec
11  /--line <= 0/
12  {
13          printf("  UDP:    %12s %12s %12s %12s %12s\n", "out(bytes)",
14              "outErrors", "in(bytes)", "inErrors", "noPort");
15          line = LINES;
16  }
17
18  mib:::udp*InDatagrams   { @in = sum(arg0);      }
19  mib:::udp*OutDatagrams  { @out = sum(arg0);     }
20  mib:::udpInErrors       { @inErr = sum(arg0);   }
21  mib:::udpInCksumErrs    { @inErr = sum(arg0);   }
22  mib:::udpOutErrors      { @outErr = sum(arg0);  }
23  mib:::udpNoPorts        { @noPort = sum(arg0);  }
24
25  profile:::tick-1sec
26  {
27          printa("          %@12d %@12d %@12d %@12d %@12d\n",
28              @out, @outErr, @in, @inErr, @noPort);
29          clear(@out); clear(@outErr); clear(@in); clear(@inErr); clear(@noPort);
30  }
```

Script udpstat.d

Line 29 uses a multiple aggregation `printa()` to generate the output. If none of those aggregations contained data at this point, no output will be generated: `printa()` skips printing when all of its aggregations arguments are empty. Once some UDP events have occurred, the aggregations are cleared on line 29—and not truncated—so that they still contain data (albeit zero), which ensures that `printa()` will print something out (and then continue to do so every second), even if that is entirely zeros.

Example

This example output caught a Web browser loading a Web site and the UDP-based DNS queries that were performed:

```
solaris# udpstat.d
  UDP:      out(bytes)    outErrors    in(bytes)     inErrors       noPort
                     0            0            1            0            0
                     6            0            6            0            0
                     2            0            2            0            0
                     0            0            0            0            0
                     0            0            0            0            0
                     4            0            4            0            0
                     0            0            1            0            0
                     0            0            0            0            0
[...]
```

udpio.d

The `udpio.d` script demonstrates the udp provider send and receive probes.

Script

This D script is about as simple as they come: probes and `printf()` statements.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-3s  %15s:%-5s      %15s:%-5s  %6s\n", "CPU",
            "LADDR", "PORT", "RADDR", "PORT", "IPLEN");
}

udp:::send
{
        printf("%-3d  %15s:%-5d  ->  %15s:%-5d  %6d\n", cpu,
            args[2]->ip_saddr, args[4]->udp_sport,
            args[2]->ip_daddr, args[4]->udp_dport, args[2]->ip_plength);
}

udp:::receive
{
        printf("%-3d  %15s:%-5d  <-  %15s:%-5d  %6d\n", cpu,
            args[2]->ip_daddr, args[4]->udp_dport,
            args[2]->ip_saddr, args[4]->udp_sport, args[2]->ip_plength);
}
```

Script udpio.d

Example

This system has an older prototype of the UDP provider. To get this script to work, the arguments had to be adjusted (`args[1]` instead of `args[2]` and `args[2]` instead of `args[4]`). A `dtrace -lvn udp:::` will show what version of the provider you have (if you have the udp provider):

```
solaris# udpio.d
CPU            LADDR:PORT                 RADDR:PORT    IPLEN
0       192.168.2.145:48912  ->      192.168.1.5:53        37
8       192.168.2.145:48912  <-      192.168.1.5:53       209
0       192.168.2.145:62535  ->      192.168.1.5:53        37
8       192.168.2.145:62535  <-      192.168.1.5:53       209
0     255.255.255.255:67     <-          0.0.0.0:68       308
7     255.255.255.255:67     <-          0.0.0.0:68       308
8     255.255.255.255:67     <-          0.0.0.0:68       308
8       192.168.2.145:34032  <-      192.168.1.5:53       117
8       192.168.2.145:58650  <-      192.168.1.5:53       102
8       192.168.2.145:62397  <-      192.168.1.5:53        96
12      192.168.2.145:34032  ->      192.168.1.5:53        42
12      192.168.2.145:58650  ->      192.168.1.5:53        58
12      192.168.2.145:62397  ->      192.168.1.5:53        55
8       192.168.3.255:137    <-    192.168.1.137:59351     58
8       192.168.3.255:137    <-    192.168.1.137:52788     58
8       192.168.3.255:137    <-    192.168.1.137:59351     58
8       192.168.3.255:137    <-    192.168.1.137:52788     58
8       192.168.3.255:137    <-    192.168.1.137:59351     58
8       192.168.3.255:137    <-    192.168.1.137:52788     58
^C
```

Various UDP packets were observed, beginning with a DNS query to host 192.168.1.5 port 53 (DNS). The inbound UDP packets to the IPv4 broadcast address 255.255.255.255 are DHCP/BOOTP discover.

This script does support IPv6 (it's the udp provider that does), but the output can get shuffled with the longer IPv6 address names.

ICMP Scripts

The Internet Control Message Protocol (ICMP) communicates errors and control messages between hosts to serve a variety of needs of the IP protocol, including sending information about invalid routes. There is currently no stable ICMP provider; however, one is planned in the Network Providers collection. Until a stable ICMP provider exists, ICMP can be traced using the fbt provider.

icmpstat.d

The `icmpstat.d` script prints ICMP statistics every second, gathered from what is available in the mib provider.

Script

ICMP probes are identified on line 5 by matching all mib probes that are in functions beginning with `icmp_`. This works well for now; however, if ICMP is processed outside of icmp functions, this technique will not match them. The only certain way to match every mib ICMP probe is to list their probe names one by one.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

mib::icmp_*:
{
        @icmp[probename] = sum(arg0);
}

profile:::tick-1sec
{
        printf("\n%Y:\n\n", walltimestamp);
        printf("  %32s %8s\n", "STATISTIC", "VALUE");
        printa("  %32s %@8d\n", @icmp);
        trunc(@icmp);
}
```

Script icmpstat.d

Example

The first output at 03:23:38 shows the mib probes that fired during an outbound ping request; the second at 03:23:39 is an inbound ping.

```
solaris# icmpstat.d

2010 Jan  6 03:23:38:

                         STATISTIC    VALUE
                    icmpInEchoReps        1
                        icmpInMsgs        1
                  rawipInDatagrams        1
                 rawipOutDatagrams        1

2010 Jan  6 03:23:39:

                         STATISTIC    VALUE
                       icmpInEchos        1
                        icmpInMsgs        1
                   icmpOutEchoReps        1
                       icmpOutMsgs        1

2010 Jan  6 03:23:40:

                         STATISTIC    VALUE
^C
```

icmpsnoop.d

The `icmpsnoop.d` script traces ICMP events live. It uses the fbt provider, which instruments a particular operating system and version, and so this script may require modifications to match the version you are using. This script was written for OpenSolaris, circa December 2009.

Script

While an ICMP receive function exists, `icmp_inbound()`, on this version of OpenSolaris there is no equivalent for sending (no `icmp_outbound()`). To see the sent ICMP packets, the type of protocol is checked in one of the later functions in the IP code path (later so that more of the fields are populated). This function, `ip_xmit_v4()`, is bypassed for faster TCP traffic, so this script isn't tracing all sends to pick out ICMP, only the slow path ones:

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option switchrate=10hz

#define IPPROTO_ICMP            1
#define IPH_HDR_LENGTH(iph)     (((struct ip *)(iph))->ip_hl << 2)

dtrace:::BEGIN
{
        /* See RFC792 and ip_icmp.h */
        icmptype[0] = "ECHOREPLY";
        icmptype[3] = "UNREACH";
        icmpcode[3, 0] = "NET";
        icmpcode[3, 1] = "HOST";
        icmpcode[3, 2] = "PROTOCOL";
        icmpcode[3, 3] = "PORT";
        icmpcode[3, 4] = "NEEDFRAG";
        icmpcode[3, 5] = "SRCFAIL";
        icmpcode[3, 6] = "NET_UNKNOWN";
        icmpcode[3, 7] = "HOST_UNKNOWN";
        icmpcode[3, 8] = "ISOLATED";
        icmpcode[3, 9] = "NET_PROHIB";
        icmpcode[3, 10] = "HOST_PROHIB";
        icmpcode[3, 11] = "TOSNET";
        icmpcode[3, 12] = "TOSHOST";
        icmpcode[3, 13] = "FILTER_PROHIB";
        icmpcode[3, 14] = "HOST_PRECEDENCE";
        icmpcode[3, 15] = "PRECEDENCE_CUTOFF";
        icmptype[4] = "SOURCEQUENCH";
        icmptype[5] = "REDIRECT";
        icmpcode[5, 0] = "NET";
        icmpcode[5, 0] = "HOST";
        icmpcode[5, 0] = "TOSNET";
        icmpcode[5, 0] = "TOSHOST";
        icmptype[8] = "ECHO";
        icmptype[9] = "ROUTERADVERT";
        icmpcode[9, 0] = "COMMON";
        icmpcode[9, 16] = "NOCOMMON";
        icmptype[10] = "ROUTERSOLICIT";
        icmptype[11] = "TIMXCEED";
        icmpcode[11, 0] = "INTRANS";
        icmpcode[11, 1] = "REASS";
        icmptype[12] = "PARAMPROB";
        icmpcode[12, 1] = "OPTABSENT";
        icmpcode[12, 2] = "BADLENGTH";
        icmptype[13] = "TSTAMP";
        icmptype[14] = "TSTAMPREPLY";
        icmptype[15] = "IREQ";
```

continues

```
        icmptype[16] = "IREQREPLY";
        icmptype[17] = "MASKREQ";
        icmptype[18] = "MASKREPLY";

        printf("%-20s  %-12s %1s %-15s %-15s %s\n", "TIME", "PROCESS", "D",
            "REMOTE", "TYPE", "CODE");
}

fbt::icmp_inbound:entry
{
        this->mp = args[1];
        this->ipha = (ipha_t *)this->mp->b_rptr;
        /* stringify manually if inet_ntoa() unavailable */
        this->addr = inet_ntoa(&this->ipha->ipha_src);
        this->dir = "<";
}

fbt::ip_xmit_v4:entry
/arg4 && args[4]->conn_ulp == IPPROTO_ICMP/
{
        this->mp = args[0];
        this->ipha = (ipha_t *)this->mp->b_rptr;
        /* stringify manually if inet_ntoa() unavailable */
        this->addr = inet_ntoa(&this->ipha->ipha_dst);
        this->dir = ">";
}

fbt::icmp_inbound:entry,
fbt::ip_xmit_v4:entry
/this->dir != NULL/
{
        this->iph_hdr_length = IPH_HDR_LENGTH(this->ipha);
        this->icmph = (icmph_t *)&this->mp->b_rptr[(char)this->iph_hdr_length];
        this->type = this->icmph->icmph_type;
        this->code = this->icmph->icmph_code;
        this->typestr = icmptype[this->type] != NULL ?
            icmptype[this->type] : lltostr(this->type);
        this->codestr = icmpcode[this->type, this->code] != NULL ?
            icmpcode[this->type, this->code] : lltostr(this->code);

        printf("%-20Y  %-12.12s %1s %-15s %-15s %s\n", walltimestamp, execname,
            this->dir, this->addr, this->typestr, this->codestr);
}
```

Script icmpsnoop.d

Most of this script is the `icmptype` and `icmpcode` associative arrays. Once a stable ICMP provider exists, these should be part of a translator in `/usr/lib/dtrace/icmp.d`, which will also pick out IP addresses and other fields of interest, making the `icmpsnoop.d` script much, much shorter (probably fewer than ten lines).

An icmp provider will also make this script much more stable. This script currently traces the IP implementation using fbt, and the IP implementation changes fairly frequently. This script will require regular maintenance to keep it working after software updates.

Example

Here the `icmpsnoop.d` script picked up various ICMP packets that were received and sent on any of the interfaces on the system:

```
solaris# icmpsnoop.d
TIME                  PROCESS       D REMOTE          TYPE            CODE
2010 Jan 16 08:29:18  ping          > 192.168.1.3     ECHO            0
2010 Jan 16 08:29:18  sched         < 192.168.1.3     ECHOREPLY       0
2010 Jan 16 08:29:19  ping          > 192.168.1.3     ECHO            0
2010 Jan 16 08:29:19  sched         < 192.168.1.3     ECHOREPLY       0
2010 Jan 16 08:29:21  sched         < 10.1.2.3        UNREACH         HOST
2010 Jan 16 08:29:22  sched         < 10.1.2.3        UNREACH         HOST
2010 Jan 16 08:29:25  in.routed     < 10.1.2.3        ECHOREPLY       0
2010 Jan 16 08:29:25  in.routed     < 10.1.2.3        ECHOREPLY       0
2010 Jan 16 08:29:25  in.routed     < 10.1.2.3        ECHOREPLY       0
2010 Jan 16 08:29:25  in.routed     < 10.1.2.3        ECHOREPLY       0
2010 Jan 16 08:29:25  in.routed     < 10.1.2.3        ECHOREPLY       0
2010 Jan 16 08:29:27  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:27  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:27  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:27  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:27  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:30  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:30  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:30  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:30  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:30  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:33  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:33  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:33  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:33  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:33  in.routed     > 224.0.0.2       ROUTERSOLICIT   0
2010 Jan 16 08:29:51  sched         < 192.168.1.109   ECHO            0
2010 Jan 16 08:29:51  sched         < 192.168.1.109   ECHOREPLY       0
2010 Jan 16 08:40:39  ping          < 127.0.0.1       ECHO            0
2010 Jan 16 08:40:39  ping          < 127.0.0.1       ECHOREPLY       0
```

Note that in the final two lines for a loopback ping, only inbound packets were observed. This is one peculiarity of loopback tracing: The kernel will skip code paths, because it knows it can deliver locally.

There are a few advantages of using DTrace for this data, instead of a packet sniffer like `tcpdump(1)` or `snoop(1M)`.

- Trace across all network interfaces simultaneously, including loopback.
- Process name available for sent packets.
- Output can be customized.
- Packet sniffers trace all packets sent and received (and all packets seen by the interface when using promiscuous mode), which can adversely affect performance.

superping.d

The `superping.d` script provides a closer measure of network round-trip time (latency) for ICMP echo request/reply. It does this by piggybacking on the `ping` command and measuring packet times within the network stack and by doing so excluding the extra time spent context switching and thread scheduling the ping application. This extra time is normally included in the times that `ping` reports, which can lead to erratic results:

```
solaris# ping -ns manta
PING manta (192.168.1.188): 56 data bytes
64 bytes from 192.168.1.188: icmp_seq=0. time=1.040 ms
64 bytes from 192.168.1.188: icmp_seq=1. time=0.235 ms
64 bytes from 192.168.1.188: icmp_seq=2. time=0.950 ms
64 bytes from 192.168.1.188: icmp_seq=3. time=0.249 ms
64 bytes from 192.168.1.188: icmp_seq=4. time=0.236 ms
[...]
```

(This is the Solaris version of `ping`, which requires `-s` for it to continually ping the target.)

manta is a nearby host; is the network latency really between 0.2 ms and 1.0 ms?

DTrace can be used to check how ping gets its times. It's likely to be calling one of the standard system time functions, like `gethrtime()` or `gettimeofday()`:

```
solaris# dtrace -x switchrate=10hz -n 'BEGIN { self->last = timestamp; }
pid$target::gettimeofday:entry { trace(timestamp - self->last); ustack();
self->last = timestamp; }' -c 'ping -ns manta'
dtrace: description 'BEGIN ' matched 3 probes
PING manta (192.168.1.188): 56 data bytes
[...]

64 bytes from 192.168.1.188: icmp_seq=3. time=0.184 ms
  8  95992              gettimeofday:entry         999760002
              libc.so.1`gettimeofday
              ping`pinger+0x140
              ping`send_scheduled_probe+0x1a1
              ping`sigalrm_handler+0x2a
              libc.so.1`__sighndlr+0x15
              libc.so.1`call_user_handler+0x2af
              libc.so.1`sigacthandler+0xdf
              libc.so.1`__pollsys+0x7
              libc.so.1`pselect+0x199
              libc.so.1`select+0x78
              ping`recv_icmp_packet+0xec
              ping`main+0x947
              ping`_start+0x7d

  8  95992              gettimeofday:entry            184488
              libc.so.1`gettimeofday
              ping`check_reply+0x27
              ping`recv_icmp_packet+0x216
              ping`main+0x947
              ping`_start+0x7d
[...]
```

Here the output from ping and DTrace are mixed. The ping output claimed the ICMP echo time was 0.184 ms; this was measured from DTrace as 184488 ns, which is consistent (the 999760002 ns time was the pause time between pings). We can see the user stack trace that led to ping calling gettimeofday().

The point here is that ping isn't measuring the time from when the ICMP echo request left the interface to when an ICMP echo reply arrived; rather, ping is measuring the time from when it sent the ICMP packet on the TCP/IP stack to when the ping command was rescheduled and put back on-CPU to receive the reply.

Script

This is a simple script based on the mib provider on Solaris, which works on the following assumption: The first ICMP packet received after ping sends an ICMP packet must be the reply (it doesn't check to confirm):

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

mib:::rawipOutDatagrams
/pid == $target/
{
        start = timestamp;
}

mib:::icmpInEchoReps
/start/
{
        this->delta = (timestamp - start) / 1000;
        printf("dtrace measured: %d us\n", this->delta);
        @a["\n  ICMP packet delta average (us):"] = avg(this->delta);
        @q["\n  ICMP packet delta distribution (us):"] =
            lquantize(this->delta, 0, 1000000, 100);
        start = 0;
}
```

Script superping.d

The script does not explicitly print out the @a and @q aggregations; that will happen automatically when the script ends.

Example

A host on the local LAN was pinged. Here, ping is executed standalone for comparison:

```
solaris# ping -ns manta 10 10
PING manta (192.168.1.188): 10 data bytes
18 bytes from 192.168.1.188: icmp_seq=0. time=0.243 ms
```

continues

```
18 bytes from 192.168.1.188: icmp_seq=1. time=0.268 ms
18 bytes from 192.168.1.188: icmp_seq=2. time=0.275 ms
18 bytes from 192.168.1.188: icmp_seq=3. time=0.308 ms
18 bytes from 192.168.1.188: icmp_seq=4. time=0.278 ms
18 bytes from 192.168.1.188: icmp_seq=5. time=0.268 ms
18 bytes from 192.168.1.188: icmp_seq=6. time=0.345 ms
18 bytes from 192.168.1.188: icmp_seq=7. time=0.241 ms
18 bytes from 192.168.1.188: icmp_seq=8. time=0.243 ms
18 bytes from 192.168.1.188: icmp_seq=9. time=0.205 ms

----manta PING Statistics----
10 packets transmitted, 10 packets received, 0% packet loss
round-trip (ms)  min/avg/max/stddev = 0.205/0.267/0.345/0.039
```

`ping` reported that the average round-trip time (on the last line) was 267 us.

Now the `superping.d` script is used. The `-c` option runs the `ping` command and provides the process ID as `$target` for use on line 7:

```
solaris# superping.d -c 'ping -s manta 56 8'
PING manta: 56 data bytes
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=0. time=0.571 ms
dtrace measured: 192 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=1. time=0.232 ms
dtrace measured: 190 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=2. time=0.475 ms
dtrace measured: 144 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=3. time=0.499 ms
dtrace measured: 155 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=4. time=0.535 ms
dtrace measured: 150 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=5. time=0.554 ms
dtrace measured: 194 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=6. time=0.513 ms
dtrace measured: 187 us
64 bytes from manta.sf.fishpong.com (192.168.1.188): icmp_seq=7. time=0.436 ms

----manta PING Statistics----
8 packets transmitted, 8 packets received, 0% packet loss
round-trip (ms)  min/avg/max/stddev = 0.232/0.477/0.571/0.108
dtrace measured: 108 us


  ICMP packet delta average (us):                              165

  ICMP packet delta distribution (us):
           value  ------------- Distribution ------------- count
               0 |                                         0
             100 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 8
             200 |                                         0
```

Disregard the output from ping in this case, which is mixed up with the output of DTrace. `ping` also now reports the average time as 477 us, because DTrace is adding to the overhead; this is why we ran `ping` standalone earlier.

`superping.d` showed that the average time from outbound ICMP to inbound ICMP was 124 us. The `ping` command earlier showed the average was 267 us (see the last line beginning in `round-trip`). This means there is about 100 us of overhead included in the time as reported by ping.

To get a more accurate reading, the commands were repeated with a count of 1,000 echo requests, instead of 10. Standalone `ping` averaged 697 us:

```
----manta PING Statistics----
1000 packets transmitted, 1000 packets received, 0% packet loss
round-trip (ms)  min/avg/max/stddev = 0.088/0.697/106.262/4.606
superping.d averaged 115 us:
  ICMP packet delta average (us):                              115

  ICMP packet delta distribution (us):
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@                             307
             100 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             690
             200 |                                         3
             300 |                                         0
```

The `ping`'s average time of 697 us was inflated by a few long responses; the maximum shown in the summary was 106.262 ms. This long latency is likely because of higher-priority threads running on all CPUs, leaving the `ping` command waiting on a dispatcher queue. DTrace can trace dispatcher queue activity if desired.

XDR Scripts

As shown in the seven-layer OSI model in Figure 6-1, a Presentation layer exists between Session and Application. In this book, there are many examples of Session tracing (sockets) and Application tracing (NFS, CIFS, and so on). This section shows the Presentation layer tracing of XDR using the (unstable) fbt provider. XDR is used in particular by NFS/RPC.

xdrshow.d

The `xdrshow.d` script counts XDR function calls on Solaris and shows the process name along with the calling function from the kernel.

Script

On this version of Solaris, XDR function calls begin with `xdr_`, which makes matching them in a probe definition easy:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
```

continues

```
4
5   dtrace:::BEGIN
6   {
7           printf("Tracing XDR calls... Hit Ctrl-C to end.\n");
8   }
9
10  fbt::xdr_*:entry
11  {
12          @num[execname, func(caller), probefunc] = count();
13  }
14
15  dtrace:::END
16  {
17          printf(" %-12s %-28s %-25s %9s\n", "PROCESS", "CALLER", "XDR_FUNCTION",
18              "COUNT");
19          printa(" %-12.12s %-28a %-25s %@9d\n", @num);
20  }
```

Script xdrshow.d

This script demonstrates `func(caller)` on line 12, which returns the kernel function name for the function that called the current one (next function on the stack).

Example

A client performed a streaming read over NFSv3, while `xdrshow.d` traced which XDR functions were called and by whom:

```
server# xdrshow.d
Tracing XDR calls... Hit Ctrl-C to end.
^C
 PROCESS      CALLER                       XDR_FUNCTION                  COUNT
 nfsd         nfs`xdr_READ3res             xdr_bytes                         1
 nfsd         genunix`xdr_bytes            xdr_opaque                        1
 nfsd         genunix`xdr_bytes            xdr_u_int                         1
 nfsd         rpcmod`svc_cots_krecv        xdr_callmsg                   13802
 nfsd         nfs`xdr_nfs_fh3_server       xdr_decode_nfs_fh3            13802
 nfsd         nfs`xdr_READ3args            xdr_nfs_fh3_server            13802
 nfsd         rpcsec`svc_authany_wrap      xdr_READ3args                 13802
 nfsd         nfs`xdr_decode_nfs_fh3       xdr_inline_decode_nfs_fh3     13803
 nfsd         rpcmod`xdr_replymsg          xdr_void                      13816
 nfsd         rpcmod`svc_cots_ksend        xdr_replymsg                  13816
 nfsd         nfs`xdr_post_op_attr         xdr_bool                      13816
 nfsd         nfs`xdr_post_op_attr         xdr_fattr3                    13816
 nfsd         nfs`xdr_READ3res             xdr_enum                      13816
 nfsd         nfs`xdr_READ3res             xdr_post_op_attr              13816
 nfsd         rpcsec`svc_authany_wrap      xdr_READ3res                  13816
 nfsd         genunix`xdr_enum             xdr_int                       13816
 nfsd         rpcmod`svc_cots_kfreeargs    xdr_READ3args                 13817
 nfsd         nfs`xdr_READ3args            xdr_nfs_fh3                   13817
 nfsd         nfs`xdr_READ3res             xdr_bool                      13817
 nfsd         nfs`xdr_READ3res             xdr_u_int                     13818
 nfsd         nfs`xdr_READ3args            xdr_u_int                     27620
 nfsd         nfs`xdr_READ3args            xdr_u_longlong_t              27620
```

At the bottom of the output we see unsigned int (`xdr_u_int`) and unsigned longlong (`xdr_u_longlong_t`) XDR functions being called by nfs `READ3args` (request arguments) and READ3res (completion result), to process the request and completion of the NFS reads. They were called 27,620 times during this trace.

This example was necessary to show that DTrace can see every layer of the OSI network model (and the TCP/IP model). But is this data useful at all?

What caught my eye in the previous output was xdr_fattr3(). This sounds like something to do with file attributes. We know the workload is performing streaming reads from large files, so why are file attribute operations so frequent? One reason could be the updating of file access time stamps; however, they have been disabled on this file system.

Reading the source code confirmed that xdr_fattr3() processed file attributes. A quick check with snoop showed that these are part of the NFS protocol:

```
solaris# snoop -v
[...]
NFS:  ----- Sun NFS -----
NFS:
NFS:  Proc = 6 (Read from file)
NFS:  Status = 0 (OK)
NFS:  Post-operation attributes:
NFS:    File type = 1 (Regular File)
NFS:    Mode = 0644
NFS:     Setuid = 0, Setgid = 0, Sticky = 0
NFS:     Owner's permissions = rw-
NFS:     Group's permissions = r--
NFS:     Other's permissions = r--
NFS:    Link count = 1, User ID = 0, Group ID = 0
NFS:    File size = 10485760, Used = 10488320
NFS:    Special: Major = 4294967295, Minor = 4294967295
NFS:    File system id = 781684113449, File id = 21
NFS:    Last access time      = 16-Jan-10 03:41:29.659625951 GMT
NFS:    Modification time     = 16-Jan-10 03:41:29.715684188 GMT
NFS:    Attribute change time = 16-Jan-10 03:41:29.715684188 GMT
NFS:
NFS:  Count = 512 bytes read
NFS:  End of file = False
NFS:
```

So, XDR is encoding and decoding these file attributes for NFS, in this case, to place the attribute information in the server response to file reads. That's every server response—even though the attributes haven't changed at all. It's possible that sending this unchanged information could be avoided by changing how the NFS protocol is implemented. That would save the CPU cycles needed to calculate the postoperation attributes and the network overhead of sending them.

Although this may sound like a promising way to improve performance, DTrace analysis often unearths many potential ways to improve performance. It's important to quantify each so that you understand those that are worth investigating and those that aren't.

To check the cost of xdr_fattr3(), the worst-case workload was applied: maximum possible NFS IOPS. If the cost of xdr_fattr3() is negligible for such a workload, then it should be negligible for all workloads. xdrshow.d was run for five seconds by adding a probe definition:

```
solaris# xdrshow.d -n 'tick-5sec { exit(0); }'
Tracing XDR calls... Hit Ctrl-C to end.
 PROCESS      CALLER                      XDR_FUNCTION                 COUNT
 nfsd         rpcmod`xdr_sizeof           xdr_fattr4_fsid                  1
[...]
 nfsd         nfs`xdr_post_op_attr        xdr_bool                    901961
 nfsd         nfs`xdr_post_op_attr        xdr_fattr3                  901961
 nfsd         nfs`xdr_READ3res            xdr_enum                    901961
[...]
```

Since xdr_fattr3() is being called more frequently, it should have a greater impact on the workload. That impact can be measured through the vtimestamp variable to sum CPU cycles for the xdr_fattr3() function:

```
solaris# dtrace -n 'fbt::xdr_fattr3:entry { self->start = vtimestamp; }
 fbt::xdr_fattr3:return /self->start/ { @ = sum(vtimestamp - self->start); }
 tick-1sec { normalize(@, 1000000); printa("%@d ms", @); clear(@); }'
dtrace: description 'fbt::xdr_fattr3:entry ' matched 3 probes
CPU     ID                    FUNCTION:NAME
 11  19148                       :tick-1sec 111 ms
 11  19148                       :tick-1sec 109 ms
 11  19148                       :tick-1sec 109 ms
 11  19148                       :tick-1sec 111 ms
 11  19148                       :tick-1sec 109 ms
 11  19148                       :tick-1sec 112 ms
[...]
```

CPU time for xdr_fattr3() was about 110 ms every second. If this was a single CPU server, that would be more than 10 percent of our CPU horsepower. Since this server has 15 online CPUs, 110 ms of on-CPU time represents 0.73 percent of available CPU cycles per second. We are looking at roughly a 1 percent win for max IOPS. Workloads with fewer xdr_fattr3() calls will be less than 1 percent.

The NFS server I'm testing is a high-end system that has already been tuned for maximum performance.[24] Although 1 percent isn't much, it would be a surprise if this particular system still had 1 percent left untuned and unnoticed—and in XDR, of all places. As a final test to quantify this, we compiled a version of NFS with a tunable, which could make xdr_fattr3() return early, without processing the file attributes. Turning on and off the tunable on a live system would allow me to see the processing cost. The results were measured as ten-minute averages of NFSv3 IOPS:

24. In the past, it has generated max IOPS results that were used by Sun marketing.

With `xdr_fattr3()`: 279,662 NFS IOPS

Without `xdr_fattr3()`: 286,791 NFS IOPS

In our test, this delivered a 2.5 percent improvement to maximum NFS IOPS to a system that was already believed to be tuned to the limit.[25] The lesson here is to check everything with DTrace, even areas that may appear rudimentary such as XDR.

Ethernet Scripts

Figure 6-6 shows the lower-level network stack for Solaris, which will be used as the target OS for this Ethernet section. See Figure 6-1 for the complete diagram, and *Solaris Internals* (McDougall and Mauro, 2006), section 18.8, "Solaris Device Driver Framework," for full descriptions.

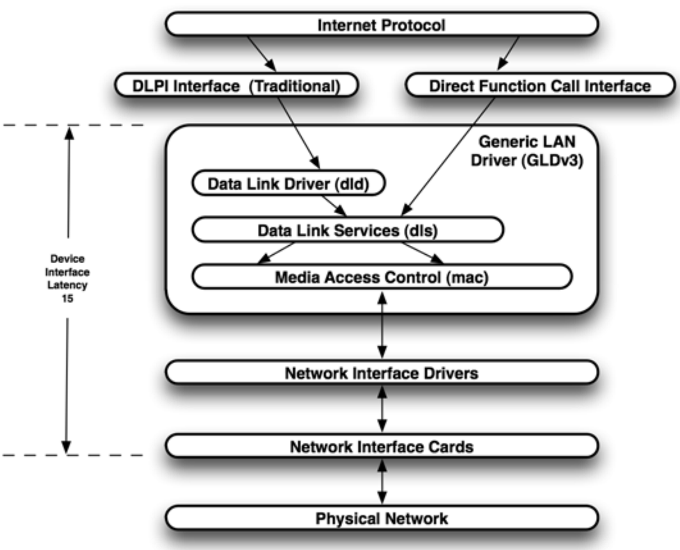


Figure 6-6 Solaris lower-level network stack

25. Although this is a possible area for improvement, to implement a fix for a 2.5 percent win for max NFS IOPS workloads only and some wins much less than 2.5 percent for lighter workloads. That's assuming that a fix can, indeed, be implemented.

Tracing Ethernet is possible at different locations in this stack, such as:

Mac: Generic layer and interface through which all Ethernet passes

Network interface (e1000g, hxge, ...): To examine specific driver internals

This section uses the fbt provider to trace mac and network driver calls.[26] Since the fbt provider examines kernel and driver source implementation, these scripts are considered unstable and will need updates as the underlying source code changes. In the future, a stable Ethernet provider may exist in mac (or higher in GLDv3) as part of the Network Providers collection, allowing stable scripts to be written. Even if these scripts do not execute, they can still be treated as examples of D programming and for the sort of data that DTrace can make available for Ethernet analysis.

Another source of network interface activity probes is the mib provider, which has interface probes placed higher in the stack. These can be useful for activity counts but can't be used to inspect Ethernet in detail.

Mac Tracing with fbt

DTrace was introduced in Solaris 10, and in the first Solaris 10 update a new network device driver architecture was introduced: GLDv3. GLDv3 provided many enhancements, including a direct function call interface for processing packets while still supporting the older, STREAMS-based interface (DLPI).[27] We start our DTrace analysis with GLDv3.

Figure 6-8 shows how GLDv3 handles both older and newer interfaces: DLPI and DFCI. Since we'd like to DTrace all I/O, tracing it further down in the stack at dls or mac should be easier; we now have one place to trace rather than two. Mac is easier because it begins to map to the GLDv3 device driver interface (DDI), which as a standard interface can make tracing simpler and more robust; the interface is less likely to change.

macops.d

The `macops.d` script traces key mac interface functions by network interface and prints a summary of their count.

---

26. This is true even if the driver is closed source.

27. The STREAMS implementation has left its mark throughout the network stack, even if it is on its way out. Many network stack functions still transfer data using STREAMS message blocks: `mblk_ts`. DTrace even has convenience functions for them: `msgsize()` and `msgdsize()`.

Script

The first argument to these mac functions is either a `mac_client_impl_t` or `mac_impl_t`, either of which can be used to retrieve the interface name (along with other interesting members). A translation table of media type number to string is declared in the `BEGIN` section:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /* See /usr/include/sys/dlpi.h */
        mediatype[0x0] = "CSMACD";
        mediatype[0x1] = "TPB";
        mediatype[0x2] = "TPR";
        mediatype[0x3] = "METRO";
        mediatype[0x4] = "ETHER";
        mediatype[0x05] = "HDLC";
        mediatype[0x06] = "CHAR";
        mediatype[0x07] = "CTCA";
        mediatype[0x08] = "FDDI";
        mediatype[0x10] = "FC";
        mediatype[0x11] = "ATM";
        mediatype[0x12] = "IPATM";
        mediatype[0x13] = "X25";
        mediatype[0x14] = "ISDN";
        mediatype[0x15] = "HIPPI";
        mediatype[0x16] = "100VG";
        mediatype[0x17] = "100VGTPR";
        mediatype[0x18] = "ETH_CSMA";
        mediatype[0x19] = "100BT";
        mediatype[0x1a] = "IB";
        mediatype[0x0a] = "FRAME";
        mediatype[0x0b] = "MPFRAME";
        mediatype[0x0c] = "ASYNC";
        mediatype[0x0d] = "IPX25";
        mediatype[0x0e] = "LOOP";
        mediatype[0x09] = "OTHER";

        printf("Tracing MAC calls... Hit Ctrl-C to end.\n");
}

/* the following are not complete lists of mac functions; add as needed */

/* mac functions with mac_client_impl_t as the first arg */
fbt::mac_promisc_add:entry,
fbt::mac_promisc_remove:entry,
fbt::mac_multicast_add:entry,
fbt::mac_multicast_remove:entry,
fbt::mac_unicast_add:entry,
fbt::mac_unicast_remove:entry,
fbt::mac_tx:entry
{
        this->macp = (mac_client_impl_t *)arg0;
        this->name = stringof(this->macp->mci_name);
        this->media = this->macp->mci_mip->mi_info.mi_media;
        this->type = mediatype[this->media] != NULL ?
```

continues

```
53                  mediatype[this->media] : lltostr(this->media);
54          this->dir = probefunc == "mac_tx" ? "->" : ".";
55          @[this->name, this->type, probefunc, this->dir] = count();
56  }
57
58  /* mac functions with mac_impl_t as the first arg */
59  fbt::mac_stop:entry,
60  fbt::mac_start:entry,
61  fbt::mac_stat_get:entry,
62  fbt::mac_ioctl:entry,
63  fbt::mac_capab_get:entry,
64  fbt::mac_set_prop:entry,
65  fbt::mac_get_prop:entry,
66  fbt::mac_rx:entry
67  {
68          this->mip = (mac_impl_t *)arg0;
69          this->name = stringof(this->mip->mi_name);
70          this->media = this->mip->mi_info.mi_media;
71          this->type = mediatype[this->media] != NULL ?
72              mediatype[this->media] : lltostr(this->media);
73          this->dir = probefunc == "mac_rx" ? "<-" : ".";
74          @[this->name, this->type, probefunc, this->dir] = count();
75  }
76
77  dtrace:::END
78  {
79          printf("  %-16s %-16s %-16s %-4s %14s\n", "INT", "MEDIA", "MAC",
80              "DATA", "CALLS");
81          printa("  %-16s %-16s %-16s %-4s %@14d\n", @);
82  }
```

Script macops.d

The mediatype table on lines 8 to 33 could be trimmed to only the types expected, such as ETHER and IB (the full list is not supported on current GLDv3 anyway).

Example

The `macops.d` script was executed for several seconds:

```
solaris# macops.d
Tracing MAC calls... Hit Ctrl-C to end.
^C
  INT              MEDIA            MAC              DATA          CALLS
  nxge1            ETHER            mac_rx           <-                9
  nxge5            ETHER            mac_rx           <-                9
  nge0             ETHER            mac_tx           ->               64
  nge0             ETHER            mac_rx           <-               72
  nge0             ETHER            mac_stat_get     .               100
  nge1             ETHER            mac_stat_get     .               100
  nge2             ETHER            mac_stat_get     .               100
  nge3             ETHER            mac_stat_get     .               100
  nxge0            ETHER            mac_stat_get     .               100
  nxge1            ETHER            mac_stat_get     .               100
  nxge4            ETHER            mac_stat_get     .               100
  nxge5            ETHER            mac_stat_get     .               100
  e1000g0          ETHER            mac_tx           ->              103
```

The output includes 64 transmit calls on `nge0` (`mac_tx()`) and 103 transmit calls on e1000g0. e1000g0? We didn't know this system had one. Checking what it is for yields the following:

```
solaris# ifconfig e1000g0
ifconfig: status: SIOCGLIFFLAGS: e1000g0: no such interface
```

This is strange; e1000g0 isn't configured, yet it is transmitting packets. Digging deeper yields the following:

```
solaris# dtrace -n 'fbt::e1000g_m_tx:entry { @[execname, stack(), ustack()] = count(); }'
dtrace: description 'fbt::e1000g_m_tx:entry ' matched 1 probe
^C

  akd
              mac`mac_tx+0x2c4
              dld`proto_unitdata_req+0x1ca
              dld`dld_wput+0x14d
              unix`putnext+0x21e
              genunix`strput+0x19d
              genunix`strputmsg+0x29a
              genunix`msgio32+0x202
              genunix`putmsg32+0x78
              unix`sys_syscall32+0x101

              libc.so.1`__putmsg+0x7
              libdlpi.so.1`i_dlpi_strputmsg+0x62
              libdlpi.so.1`dlpi_send+0x124
              libak.so.1`ak_ciodlpi_transmit+0x8b
              libak.so.1`ak_cio_link_transmit_one+0x32
              libak.so.1`ak_cio_link_tx+0x225
              libak.so.1`ak_thread_start+0x7d
              libc.so.1`_thrp_setup+0x9b
              libc.so.1`_lwp_start
               27
```

Oh...*that* e1000g! (We had forgotten this system had a cluster card interconnect that uses e1000g, which the akd process manages.) The previous one-liner checked the process name, kernel, and user stack traces for `e1000g_m_tx()`. The process name and user stack trace will often be invalid because TCP buffering of sends; however, the previous stack appears to have caught it correctly (since the send went straight from the system call).

Network Device Driver Tracing with fbt

The fbt provider can examine the operation of network device drivers. Here we examine the nge driver (Nvidia Gigabit Ethernet) as an example of what can be done.

We demonstrate two approaches: tracing the internal operation of the driver with whatever functions the programmer chose to write and tracing the interface to GLDv3, which is well defined and common to other drivers.

Driver Internals

You can start examining the internal execution of the device driver by performing lists of probes and frequency counts. Listing probes for the nge driver (on Solaris) yields the following:

```
solaris# dtrace -ln 'fbt:nge::entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
    4        fbt               nge                 nge_set_loop_mode entry
    6        fbt               nge                nge_fini_send_ring entry
    8        fbt               nge                nge_init_send_ring entry
   10        fbt               nge              nge_reinit_send_ring entry
   12        fbt               nge                nge_init_recv_ring entry
   14        fbt               nge              nge_reinit_recv_ring entry
   16        fbt               nge                nge_fini_buff_ring entry
   18        fbt               nge                nge_init_buff_ring entry
   20        fbt               nge              nge_reinit_buff_ring entry
[...]
solaris# dtrace -ln 'fbt:nge::entry' | wc
     145     725   10874
```

This shows that there are 144 functions in nge available to probe. Instead of turning to the source code right now (if available), we can narrow down this list to probes of interest, such as the send/receive probes.

Sometimes guesswork pays off. What would the programmer have called the send and receive functions?

```
solaris# dtrace -ln 'fbt:nge::entry' | egrep 'send|receive|recv|read|write|tx|rx'
91777        fbt               nge                nge_fini_send_ring entry
91779        fbt               nge                nge_init_send_ring entry
91781        fbt               nge              nge_reinit_send_ring entry
91783        fbt               nge                nge_init_recv_ring entry
91785        fbt               nge              nge_reinit_recv_ring entry
91857        fbt               nge                      nge_rx_setup entry
91859        fbt               nge                      nge_tx_setup entry
91901        fbt               nge                   nge_recv_packet entry
91903        fbt               nge                  nge_rxsta_handle entry
91905        fbt               nge                     nge_recv_ring entry
91907        fbt               nge                   nge_tx_dmah_pop entry
91909        fbt               nge                  nge_tx_dmah_push entry
91911        fbt               nge                  nge_tx_desc_sync entry
91913        fbt               nge                      nge_tx_alloc entry
91915        fbt               nge                      nge_tx_start entry
91917        fbt               nge                     nge_send_copy entry
91919        fbt               nge                   nge_send_mapped entry
91921        fbt               nge                          nge_send entry
91943        fbt               nge                nge_tx_recycle_all entry
91951        fbt               nge                 nge_sum_rxd_check entry
91953        fbt               nge                 nge_sum_txd_check entry
```

```
  91971       fbt          nge                nge_tx_recycle entry
  91989       fbt          nge             nge_hot_rxd_check entry
  91991       fbt          nge             nge_hot_txd_check entry
  91999       fbt          nge              nge_sum_rxd_fill entry
  92001       fbt          nge              nge_sum_txd_fill entry
  92005       fbt          nge                   nge_receive entry
  92035       fbt          nge              nge_hot_rxd_fill entry
  92037       fbt          nge              nge_hot_txd_fill entry
  92049       fbt          nge                      nge_m_tx entry
  92053       fbt          nge              nge_recv_recycle entry
```

We used the `egrep(1)` command to search for various likely terms. Here it has matched functions from nge with promising names such as `nge_recv_packet()`, `nge_send()`, and `nge_receive()`. (We happen to know that `nge_m_tx()` is also promising, because ***drivername***`_m_tx()` has become the convention for the mac interface transmit function.)

Performing a frequency count test of all that are called is another way to narrow down potential probes of interest:

```
solaris# dtrace -n 'fbt:nge::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:nge::entry ' matched 144 probes
cccccccccccccccccc^C

  nge_tx_recycle                                                    1
  nge_hot_txd_check                                                 6
  nge_atomic_shl32                                                 12
  nge_check_copper                                                 12
  nge_chip_cyclic                                                  12
  nge_chip_factotum                                                12
  nge_factotum_link_check                                          12
  nge_factotum_stall_check                                         12
  nge_interrupt_optimize                                           12
  nge_wake_factotum                                                12
  nge_hot_txd_fill                                                 17
  nge_send                                                         17
  nge_tx_alloc                                                     17
  nge_tx_start                                                     17
  nge_m_tx                                                         18
  nge_tx_desc_sync                                                 18
  nge_hot_rxd_fill                                                 39
  nge_intr_handle                                                  54
  nge_receive                                                      54
  nge_recv_ring                                                    54
  nge_reg_put32                                                    83
  nge_reg_put16                                                    90
  nge_hot_rxd_check                                                93
  nge_chip_intr                                                    94
  nge_reg_get8                                                    180
  nge_reg_put8                                                    180
  nge_m_stat                                                      228
  nge_reg_get32                                                   275
  nge_reg_get16                                                   577
```

Although this one-liner was running, the `c` character was typed 17 times to cause some network read and writes (this was performed over an `ssh` session).

This showed that nge_send() fired 17 times, and nge_receive() fired 54 times: This rate coincides with what we saw with other system tools (netstat -i). Note that nge_recv_packet() never fired, so we can drop that from our investigation.

Checking how nge_send() and nge_receive() are called to get a better insight into the driver yields the following:

```
solaris# dtrace -n 'fbt::nge_send:entry,fbt::nge_receive:entry
{ @[probefunc,  stack()] = count(); }'
dtrace: description 'fbt::nge_send:entry,fbt::nge_receive:entry ' matched 2 probes
^C
[...]
  nge_send
              nge`nge_m_tx+0x60
              mac`mac_tx+0x2c4
              dld`str_mdata_fastpath_put+0xa4
              ip`tcp_send_data+0x94e
              ip`tcp_output+0x7fa
              ip`squeue_enter+0x330
              ip`tcp_sendmsg+0xfd
              sockfs`so_sendmsg+0x1c7
              sockfs`socket_sendmsg+0x61
              sockfs`socket_vop_write+0x63
              genunix`fop_write+0xa4
              genunix`write+0x2e2
              genunix`write32+0x22
              unix`sys_syscall32+0x101
                5
  nge_receive
              nge`nge_intr_handle+0xd5
              nge`nge_chip_intr+0x81
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
               32
```

nge_send() is called from nge_m_tx(), which was called by mac_tx(). That's a function from the mac layer that was traced earlier. We expected this given that nge_m_tx() is the conventional name for the mac interface function; however, if we didn't know that, we would have still discovered it by examining the previous stack trace. The stacks also show that nge_receive() was called from a hardware interrupt.

The definition for nge_m_tx() can be read from uts/common/io/nge/nge_tx.c:

```
/*
 * nge_m_tx : Send a chain of packets.
 */
mblk_t *
nge_m_tx(void *arg, mblk_t *mp)
{
        nge_t *ngep = arg;
[...]
```

The first argument is an `nge_t` to describe the network interface; the second is an `mblk_t` to contain the packets to send.

Examining `nge_t` from `uts/common/io/nge/nge.h` yields the following:

```
typedef struct nge {
        /*
         * These fields are set by attach() and unchanged thereafter ...
         */
        dev_info_t              *devinfo;       /* device instance      */
        mac_handle_t            mh;             /* mac module handle    */
        chip_info_t             chipinfo;
        ddi_acc_handle_t        cfg_handle;     /* DDI I/O handle       */
        ddi_acc_handle_t        io_handle;      /* DDI I/O handle       */
        void                    *io_regs;       /* mapped registers     */
[...]
        char                    ifname[8];      /* "nge0" ... "nge999" */

        enum nge_mac_state      nge_mac_state;  /* definitions above    */
        enum nge_chip_state     nge_chip_state; /* definitions above    */
[...]
```

Some interesting structure members are apparent: `ifname` has the interface name as a string, and the `*_state` members convey different error states of the interface. For example, `ifname` could be traced this way (remember `arg0` was actually a `void *`, so it will need to be cast to a type):

```
solaris# dtrace -n 'fbt::nge_m_tx:entry
{ this->n = (nge_t *)arg0; trace(stringof(this->n->ifname)); }'
dtrace: description 'fbt::nge_m_tx:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  8    276                   nge_m_tx:entry   nge0
  0    276                   nge_m_tx:entry   nge0
  0    276                   nge_m_tx:entry   nge0
  0    276                   nge_m_tx:entry   nge0
  0    276                   nge_m_tx:entry   nge0
  0    276                   nge_m_tx:entry   nge0
^C
```

The other argument to `nge_m_tx()` was an `mblk_t` pointer, and DTrace already has convenience functions for those (on Solaris):

```
solaris# dtrace -n 'fbt::nge_m_tx:entry
{ this->n = (nge_t *)arg0; printf("%s %d bytes",
stringof(this->n->ifname), msgdsize(args[1])); }'
dtrace: description 'fbt::nge_m_tx:entry ' matched 1 probe

CPU     ID                    FUNCTION:NAME
  0    276                   nge_m_tx:entry nge0 102 bytes
  5    276                   nge_m_tx:entry nge0 198 bytes
  5    276                   nge_m_tx:entry nge0 150 bytes
  6    276                   nge_m_tx:entry nge0 150 bytes
```

continues

```
  4    276                   nge_m_tx:entry nge0 290 bytes
  4    276                   nge_m_tx:entry nge0 494 bytes
  4    276                   nge_m_tx:entry nge0 102 bytes
  6    276                   nge_m_tx:entry nge0 150 bytes
  8    276                   nge_m_tx:entry nge0 42 bytes
  8    276                   nge_m_tx:entry nge0 42 bytes
  8    276                   nge_m_tx:entry nge0 66 bytes
^C
```

Now we have a one-liner that can trace sends from the network device driver, bearing in mind that each send may be a chain of packets, as noted by the `nge_m_tx()` comment shown earlier. From here, we can continue digging into nge to gather more information. By using such DTrace one-liners, we've narrowed 144 fbt provider probes down to a few of interest, which we know fire at rates that seem plausible. Examining the previous stack traces has also given us an idea of the code path.

Driver Interface

Although we can `dtrace` device drivers via their internal functions, we can also trace their operation via the mac device driver interface. This is a becoming a well-defined and documented interface.[28] It needs to be, so that third-party vendors can quickly learn and write new drivers that interface to mac.

The actual driver interface is achieved by declaring a list of driver functions that mac will call. This includes the `nge_m_tx()` function seen earlier:

```
uts/common/io/nge/nge_main.c:

static mac_callbacks_t nge_m_callbacks = {
        NGE_M_CALLBACK_FLAGS,
        nge_m_stat,
        nge_m_start,
        nge_m_stop,
        nge_m_promisc,
        nge_m_multicst,
        nge_m_unicst,
        nge_m_tx,
        nge_m_ioctl,
        nge_m_getcapab,
        NULL,
        NULL,
        nge_m_setprop,
        nge_m_getprop
};
```

This structure maps to `mac_callbacks_t`, which is the device driver interface into GLDv3 via mac:

28. See PSARC 2009/638 for the interface description.

```
typedef struct mac_callbacks_s {
        uint_t          mc_callbacks;   /* Denotes which callbacks are set */
        mac_getstat_t   mc_getstat;     /* Get the value of a statistic */
        mac_start_t     mc_start;       /* Start the device */
        mac_stop_t      mc_stop;        /* Stop the device */
        mac_setpromisc_t mc_setpromisc; /* Enable or disable promiscuous mode */
        mac_multicst_t  mc_multicst;    /* Enable or disable a multicast addr */
        mac_unicst_t    mc_unicst;      /* Set the unicast MAC address */
        mac_tx_t        mc_tx;          /* Transmit a packet */
        mac_ioctl_t     mc_ioctl;       /* Process an unknown ioctl */
        mac_getcapab_t  mc_getcapab;    /* Get capability information */
        mac_open_t      mc_open;        /* Open the device */
        mac_close_t     mc_close;       /* Close the device */
        mac_set_prop_t  mc_setprop;
        mac_get_prop_t  mc_getprop;
} mac_callbacks_t;
```

The mac function prototypes are defined in `uts/common/sys/mac_provider.h`:

```
/*
 * MAC driver entry point types.
 */
typedef int             (*mac_getstat_t)(void *, uint_t, uint64_t *);
typedef int             (*mac_start_t)(void *);
typedef void            (*mac_stop_t)(void *);
typedef int             (*mac_setpromisc_t)(void *, boolean_t);
typedef int             (*mac_multicst_t)(void *, boolean_t, const uint8_t *);
typedef int             (*mac_unicst_t)(void *, const uint8_t *);
typedef void            (*mac_ioctl_t)(void *, queue_t *, mblk_t *);
typedef void            (*mac_resources_t)(void *);
typedef mblk_t          *(*mac_tx_t)(void *, mblk_t *);
typedef boolean_t       (*mac_getcapab_t)(void *, mac_capab_t, void *);
typedef int             (*mac_open_t)(void *);
typedef void            (*mac_close_t)(void *);
typedef int             (*mac_set_prop_t)(void *, const char *, mac_prop_id_t,
                            uint_t, const void *);
typedef int             (*mac_get_prop_t)(void *, const char *, mac_prop_id_t,
                            uint_t, uint_t, void *, uint_t *);
```

Each of these arguments can also be examined using the fbt provider.

Tip

As discussed before, the fbt provider exposes the kernel source code, which is considered an unstable interface. This means scripts written for it are likely to break whenever the kernel is updated and functions change.

However, the functions listed in the `nge_m_callbacks` struct are an interface defined by GLDv3, which is followed by multiple vendors writing third-party drivers. This interface, specifically the arguments, return values, the number of functions, and their role, is therefore unlikely to change frequently.

Keep a lookout for other such interfaces; DTracing them may answer your questions and provide reasonably robust scripts, despite using the unstable fbt provider.

ngesnoop.d

This script traces nge driver send and receive and prints details from the Ethernet header. This could be a starting point for further customization, such as printing events from other areas of the system alongside Ethernet events.

This uses the fbt provider to examine the operation of nge and mac. This script will need changes to work on different versions of the nge driver and for updates to the Solaris GLD interface.

While DTrace is tracing, the interfaces are not put into promiscuous mode, because they can be with network sniffers.

Script

We selected the `nge_send()` and `mac_rx()` functions to probe for the send and receive events. The script populates two clause-local variables: `this->nge` with an `nge_t` pointer for interface information and `this->mp` for the message block pointer for frame/packet information:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-15s  %-8s %-2s %-17s  %-17s  %-5s %5s\n", "TIME(us)",
            "INT", "D", "SOURCE", "DEST", "PROTO", "BYTES");
}

fbt::nge_recv_ring:entry
{
        self->ngep = args[0];
}

fbt::mac_rx:entry
/self->ngep/
{
        this->mp = args[2];
        this->nge = self->ngep;
        this->dir = "<-";
        self->ngep = 0;
}

fbt::nge_send:entry
{
        this->nge = (nge_t *)arg0;
        this->mp = args[1];
        this->dir = "->";
}

fbt::mac_rx:entry,
fbt::nge_send:entry
/this->mp/
{
        this->eth = (struct ether_header *)this->mp->b_rptr;
        this->s = (char *)&this->eth->ether_shost;
```

```
39          this->d = (char *)&this->eth->ether_dhost;
40          this->t = ntohs(this->eth->ether_type);
41          printf("%-15d  %-8s %2s ", timestamp / 1000, this->nge->ifname,
42              this->dir);
43          printf("%02x:%02x:%02x:%02x:%02x:%02x  ", this->s[0], this->s[1],
44              this->s[2], this->s[3], this->s[4], this->s[5]);
45          printf("%02x:%02x:%02x:%02x:%02x:%02x  ", this->d[0], this->d[1],
46              this->d[2], this->d[3], this->d[4], this->d[5]);
47          printf(" %-04x %5d\n", this->t, msgdsize(this->mp));
48  }
```

Script ngesnoop.d

Lines 43 to 46 print the Ethernet addresses as a series of bytes in hexadecimal, separated by colons. Each byte is accessed using array operators, `this->s[0]` for the first byte, and so on.

Line 40 used the `ntohs()` built-in to convert endian from network to host order, and line 47 used the `msdsize()` built-in to determine the size in bytes of a message (`STREAMS`).

Example

The output shows which Ethernet frames are being processed by any nge interface on the system:

```
solaris# ngesnoop.d
TIME(us)         INT      D  SOURCE             DEST               PROTO BYTES
64244351306      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64244385653      nge0     <- 00:14:4f:ca:fb:04  ff:ff:ff:ff:ff:ff   0800   342
64244409073      nge0     <- 00:14:4f:3b:76:c8  00:14:4f:ed:d4:1c   0800    60
64244451504      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64244451616      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64244451637      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64244451711      nge0     <- 00:14:4f:3b:76:c8  00:14:4f:ed:d4:1c   0800    60
64244509203      nge0     <- 00:14:4f:3b:76:c8  00:14:4f:ed:d4:1c   0800    60
64244618499      nge0     -> 00:14:4f:ed:d4:1c  00:1b:24:93:8a:6e   0800   102
64244618645      nge0     -> 00:14:4f:ed:d4:1c  00:1b:24:93:8a:6e   0800   102
64244551065      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64244551098      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   262
64244551121      nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   262
64244551245      nge0     <- 00:14:4f:3b:76:c8  00:14:4f:ed:d4:1c   0800    60
[...]
```

The `TIME` column is printed in case the output is shuffled and requires postsorting; it is the time since boot in microseconds. It could also be examined for packet latency, by comparing the delta between two lines.

The previous `ngesnoop.d` script prints Ethernet header details including `SOURCE` and `DESTination` addresses. It may be desirable to print these differently, in terms of local and remote addresses. After customizing the script, we get this:

```
solaris# ngesnoop2.d
TIME(us)          INT       D  LOCAL              REMOTE             PROTO BYTES
64424890458       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64424946171       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800    60
64424990648       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64424990759       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64424990851       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800    60
64425081306       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   651
64425150070       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800    54
64425090329       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64425090355       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   182
64425090382       nge0     -> 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   262
64425090459       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800    60
64425136345       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800    60
64425150208       nge0     <- 00:14:4f:ed:d4:1c  00:14:4f:3b:76:c8   0800   235
[...]
```

A series of packets transmitted between two hosts is more easily identifiable as the `LOCAL` and `REMOTE` columns contain the same addresses.

ngelink.d

This script traces link status events on the nge interface, such as negotiating different Ethernet speeds.

Script

This version of the nge driver calls `nge_check_copper()` in response to interrupts, in case a state has changed. The `ngelink.d` script traces the calls to `nge_check_copper()` and prints nge state details if one of the nge properties did change:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate

int seen[nge_t *];
int up[nge_t *];
int speed[nge_t *];
int duplex[nge_t *];
int last[nge_t *];

dtrace:::BEGIN
{
        printf("%-20s  %-10s %6s %8s %8s   %s\n", "TIME", "INT", "UP",
            "SPEED", "DUPLEX", "DELTA(ms)");
}

fbt::nge_check_copper:entry
{
        self->ngep = args[0];
}
```

```
22
23  fbt::nge_check_copper:return
24  /self->ngep && (!seen[self->ngep] ||
25      (up[self->ngep] != self->ngep->param_link_up ||
26      speed[self->ngep] != self->ngep->param_link_speed ||
27      duplex[self->ngep] != self->ngep->param_link_duplex))/
28  {
29          this->delta = last[self->ngep] ? timestamp - last[self->ngep] : 0;
30          this->name = stringof(self->ngep->ifname);
31          printf("%-20Y  %-10s %6d %8d %8d   %d\n", walltimestamp, this->name,
32              self->ngep->param_link_up, self->ngep->param_link_speed,
33              self->ngep->param_link_duplex, this->delta / 1000000);
34          seen[self->ngep] = 1;
35          last[self->ngep] = timestamp;
36  }
37
38  fbt::nge_check_copper:return
39  /self->ngep/
40  {
41          up[self->ngep] = self->ngep->param_link_up;
42          speed[self->ngep] = self->ngep->param_link_speed;
43          duplex[self->ngep] = self->ngep->param_link_duplex;
44          self->ngep = 0;
45  }
```

Script ngelink.d

A `seen[]` associative array is used to show interface status when the script begins tracing. The predicate on lines 24 to 27 checks the `seen[]` array for printing the first set of output and checks for state changes for the nge properties, which are remembered in separate associative arrays.

Example

For this example, we unplugged the network cable briefly:

```
solaris# ngelink.d
TIME                  INT            UP    SPEED   DUPLEX   DELTA(ms)
2010 Jan 13 02:44:35  nge2            0        0        0   0
2010 Jan 13 02:44:35  nge0            1     1000        2   0
2010 Jan 13 02:44:35  nge3            0        0        0   0
2010 Jan 13 02:44:36  nge1            0        0        0   0
2010 Jan 13 02:45:18  nge0            0        0        0   43000
2010 Jan 13 02:45:20  nge0            1     1000        2   2109
^C
```

The `DELTA(ms)` column is showing the time between events (lines of output). The time that the nge0 interface was offline can be seen: 2.1 seconds. And the time it took us to walk to the server in our server room and unplug the cable after running the script was 43 seconds.

Common Mistakes

These are some common mistakes and sources of confusion for DTracing network I/O.

Receive Context

Receive context refers to the execution context of the system when receiving packets and how it applies to using DTrace to track received packets.

Incorrect

Let's say we'd like to trace the process name of the application that is receiving TCP packets. The mib provider allows TCP receive to be traced, so this sounds like it could be answered with a one-liner to show the current process name (`execname`):

```
solaris# dtrace -n 'mib:::tcpInDataInorderBytes { @[execname] = sum(args[0]); }'
```

A known workload was applied using the `ttcp` tool to test this one-liner. The following was run on local and remote hosts, both Solaris:

```
localhost# ttcp -r -l1024 -n10000 -s

remotehost# ttcp -t -s -l1024 -n10000 localhost < /dev/zero
```

The `ttcp` process on the localhost should receive 10,240,000 bytes of data (10,000 1KB I/O), which is our known load. Testing the one-liner yields the following:

```
solaris# dtrace -n 'mib:::tcpInDataInorderBytes { @[execname] = sum(args[0]); }'
dtrace: description 'mib:::tcpInDataInorderBytes ' matched 3 probes
^C

  sched                                                          10240048
```

This shows all packets have arrived in kernel context, `sched`, not the `ttcp` application. To illustrate why this didn't identify the application, Figure 6-7 shows how networking would need to work for this one-liner to identify the correct context.

1. The socket receive buffer is checked and found to be empty.
2. Wait for packet *while on-CPU*.
3. Receive in application context.

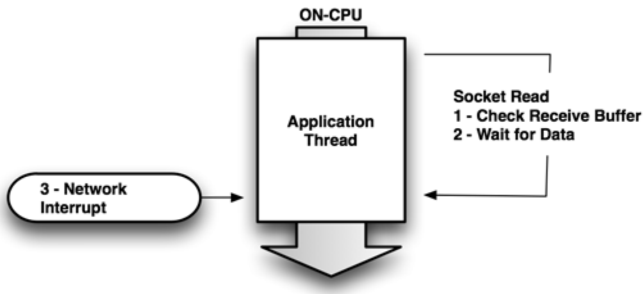


Figure 6-7 Receive context (wrong)

For the receive to occur in application context, the application thread would need to stay on-CPU while waiting for the packet. What would the thread do while it waited on-CPU? Spin loop?

This would not be an efficient operating system design. What actually happens in this case is that the application thread context-switches off-CPU to allow other threads to be run (even the idle thread).

More Accurate

Packets are received in interrupt context (kernel), processed by the TCP/IP stack, and then the application thread is context-switched back on-CPU to receive the data, as shown in Figure 6-8.

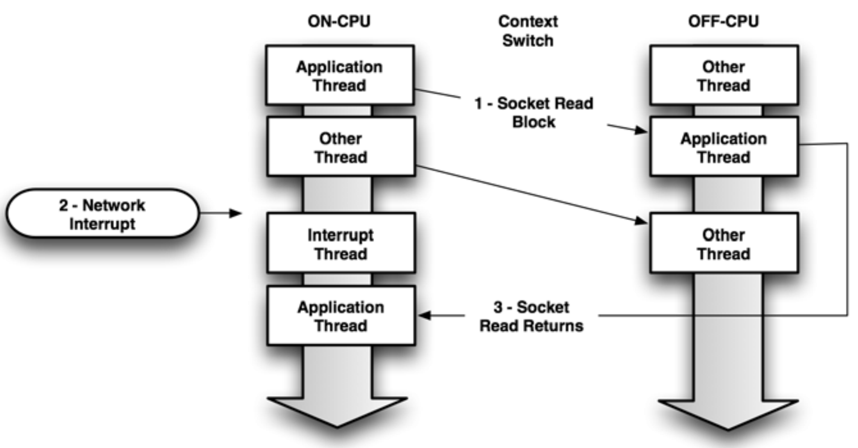


Figure 6-8 Receive context

The socket receive buffer is empty; the read will block.

Receive in network interrupt context.

Receive in application context.

Examining the kernel stacks when receiving TCP data yields the following:

```
solaris# dtrace -n 'mib:::tcpInDataInorderBytes
{ @[execname, stack()] = sum(args[0]); }'
dtrace: description 'mib:::tcpInDataInorderBytes ' matched 3 probes
^C
[...]

  sched
              ip`squeue_enter+0x330
              ip`ip_input+0xe31
              mac`mac_rx_soft_ring_process+0x184
              mac`mac_rx_srs_proto_fanout+0x46f
              mac`mac_rx_srs_drain+0x235
              mac`mac_rx_srs_process+0x1db
              mac`mac_rx_common+0x94
              mac`mac_rx+0xac
              mac`mac_rx_ring+0x4c
              igb`igb_intr_rx+0x67
              unix`av_dispatch_autovect+0x7c
              unix`dispatch_hardint+0x33
              unix`switch_sp_and_call+0x13
         10231928
```

This is similar to the stack trace we saw before for read packets, with igb as the network interface instead of nge. Further up the stack (not visible here), a context switch occurs, and the application is brought back on-CPU to receive the TCP data.

To further complicate things, there are at least two scenarios where the application context switch may not occur directly after receiving the packet.

A context switch to a different higher-priority thread may occur first.

The application thread may already be running on a different CPU, reading from the socket buffer. Instead of context switching, the network interrupt simply tops up the socket buffer.

If it isn't clear by now, DTracing TCP/IP internals is very difficult. With the future introduction of stable network providers, this should become easier.

See the "TCP Scripts" section for further tracing of receive packets.

Send Context

Send context refers to the execution context of the system when sending network packets and how it applies to using DTrace to track and observe network packet sends.

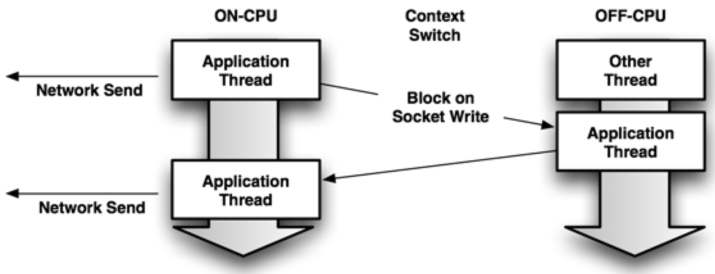


Figure 6-9 Send context (simple)

Incorrect

Tracing the process name when sending packets sounds like this could be easier than for receives, because the application thread may context switch off-CPU *after* the packet is sent. Figure 6-9 shows this assumption.

To test this, the same `ttcp` workload as before was applied, this time with DTrace on the target host to examine the process name when TCP packets are sent:

```
solaris# dtrace -qn 'mib:::tcpOutDataBytes { @[execname] = sum(args[0]); }'
^C

  ssh                                                                   48
  sshd                                                                1904
  ttcp                                                             1649664
  sched                                                            8590444
```

According to this DTrace one-liner, the `ttcp` application sent only 1,649,664 bytes, which is 16 percent of the expected value (10,240,000). `sched`, the kernel, sent 8,590,444 bytes. Summing `sched` and `ttcp` gives 10,240,108 bytes, the correct value (plus other TCP bytes from unrelated apps).

The application is still on-CPU some of the time during TCP send, but most of the time a kernel thread was doing the sending. This could mean that the requests were queued or buffered and processed later.

More Accurate

By including `stack()` in the aggregation key, we can examine the kernel stack backtraces for TCP sends and see why the kernel is performing them:

```
solaris# dtrace -n 'mib:::tcpOutDataBytes { @[execname, stack()] = sum(args[0]); }'
dtrace: description 'mib:::tcpOutDataBytes ' matched 4 probes
^C
[...]
  ttcp
              ip`tcp_wput_data+0x75a
              ip`tcp_output+0x7c5
              ip`squeue_enter+0x416
              ip`tcp_wput+0xf8
              sockfs`sostream_direct+0x168
              sockfs`socktpi_write+0x179
              genunix`fop_write+0x69
              genunix`write+0x208
              genunix`write32+0x1e
              unix`sys_syscall32+0x1fc
          1351680
  sched
              ip`tcp_wput_data+0x75a
              ip`tcp_rput_data+0x3042
              ip`squeue_enter_chain+0x2c0
              ip`ip_input+0xa42
              dls`i_dls_link_rx+0x2b9
              mac`mac_do_rx+0xba
              mac`mac_rx+0x1b
              nge`nge_receive+0x47
              nge`nge_intr_handle+0xbd
              nge`nge_chip_intr+0xca
              unix`av_dispatch_autovect+0x8c
              unix`dispatch_hardint+0x2f
              unix`switch_sp_and_call+0x13
          8839168
```

This shows that the kernel TCP send is originating from a network interrupt (nge). A packet is received, which is handed to `tcp_rput_data()`, which then calls `tcp_wput_data()` to send the next packet. Examining the `tcp_rput_data()` source shows that it checks whether this is more data to send from the *TCP window buffer* and, if so, sends it. This is shown in Figure 6-10.

Send from application context (thread1 is still on-CPU).

Fetch next data from buffer to send.

Send from network interrupt context.

On Mac OS X, the stack for kernel sends from the TCP buffer looks like this:

```
solaris# dtrace -n 'ether_frameout:entry { @[execname, stack()] = count() }'
dtrace: description 'ether_frameout:entry ' matched 1 probe
^C
[...]
  kernel_task
              mach_kernel`ifnet_input+0xe43
              mach_kernel`ifnet_output+0x4d
              mach_kernel`ip_output_list+0x1d9f
              mach_kernel`tcp_setpersist+0x16e
              mach_kernel`tcp_output+0x17ab
```

```
                mach_kernel`tcp_input+0x3848
                mach_kernel`ip_rsvp_done+0x1c6
                mach_kernel`ip_input+0x17bd
                mach_kernel`ip_input+0x17f9
                mach_kernel`proto_input+0x92
                mach_kernel`ether_detach_inet+0x1c9
                mach_kernel`ifnet_input+0x2f8
                mach_kernel`ifnet_input+0xa51
                mach_kernel`ifnet_input+0xcaf
                mach_kernel`call_continuation+0x1c
               6980
```

`tcp_input()` calls `tcp_output()`, for the same reason as on Solaris.

Packet Size

Although over-the-network interface data may be sent in packets of size 1,500 bytes or so (maximum transmission unit), the size can vary throughout the network stack. A single socket write may be split into many IP sends by the time the data is sent to the network interface driver. Since this can occur after the TCP layer, tracing tcp functions may not be a one-to-one mapping to packets; TCP could send data that was then split by IP.

Some network cards and drivers increase the size of network I/O beyond jumbo frames to improve performance. TCP Large Send Offload is an example of this, where oversized packets (more than 50KB) can be sent to the network card, which splits them into MTU-sized packets in hardware.

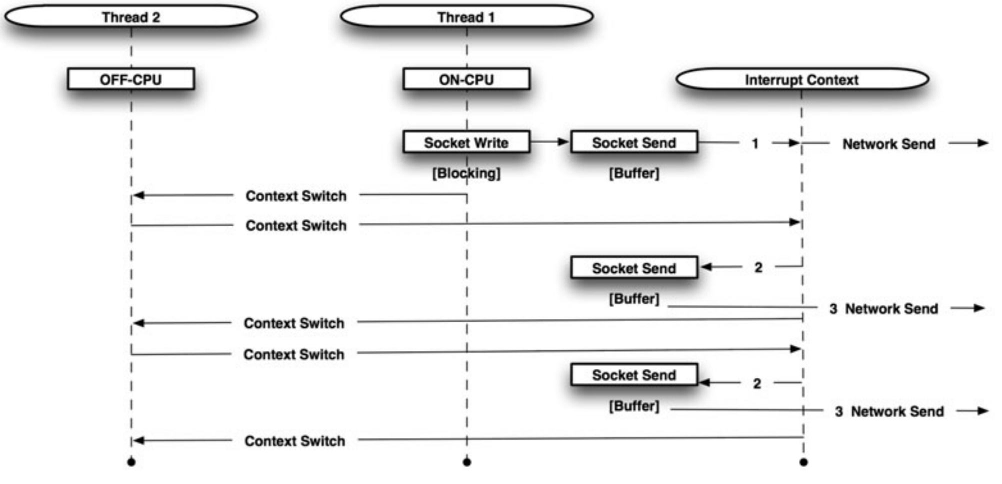


Figure 6-10 Send context (buffering)

Stack Reuse

When we find functions such as ip_output(), we may be tempted to assume that there is a one-to-one mapping of such functions to packets. The factors mentioned previously could inflate the number of actual IP packets from the observed IP and TCP function counts. However, there are cases where the actual IP packets may be half (or fewer) of what the function counts suggest.

Take a careful look at the following stack traces, aggregated on the ip:::send probe, traced on an OpenSolaris system:

```
solaris# dtrace -x stackframes=200 -n 'ip:::send { @[stack()] = count(); }'
dtrace: description 'ip:::send ' matched 4 probes
^C
              ip`ip_output+0xead
              ip`tcp_send_data+0xa13
              ip`tcp_rput_data+0x35b4
              ip`tcp_input+0x74
              ip`squeue_enter_chain+0x2e8
              ip`ip_input+0x9db
              ip`ip_rput+0x185
              unix`putnext+0x31a
              tun`tun_rdata_v4+0x642
              tun`tun_rdata+0x1a5
              tun`tun_rproc+0x139
              tun`tun_rput+0x29
              unix`putnext+0x31a
              ip`ip_fanout_proto+0xba2
              ip`ip_proto_input+0xd9c
              ip`ip_fanout_proto_again+0x375
              ip`ip_proto_input+0xbec
              ip`ip_input+0x97a
              ip`ip_rput+0x185
              unix`put+0x28c
              nattymod`natty_rput_pkt+0x3ca
              nattymod`natty_rput_other+0x103
              nattymod`natty_rput+0x37
              unix`putnext+0x31a
              ip`udp_input+0x116e
              ip`udp_input_wrapper+0x25
              ip`udp_conn_recv+0x89
              ip`ip_udp_input+0x703
              ip`ip_input+0x914
              dls`i_dls_link_rx+0x2dc
              mac`mac_rx+0x7a
              e1000g`e1000g_intr+0xf6
              unix`av_dispatch_autovect+0x97
              unix`intr_thread+0x50
                6

              ip`ip_output+0x25dd
              ip`ip_wput+0x5a
              unix`putnext+0x31a
              tun`tun_wputnext_v4+0x2bb
              tun`tun_wproc_mdata+0xca
              tun`tun_wproc+0x38
              tun`tun_wput+0x29
              unix`putnext+0x31a
              ip`ip_xmit_v4+0x786
              ip`ip_wput_ire+0x228a
```

```
ip`ip_output+0xead
ip`tcp_send_data+0xa13
ip`tcp_output+0x7d2
ip`squeue_enter+0x469
ip`tcp_wput+0xfb
sockfs`sostream_direct+0x176
sockfs`socktpi_write+0x18d
genunix`fop_write+0x43
genunix`write+0x21d
genunix`write32+0x20
unix`sys_syscall32+0x1ff
 15
```

Notice that the `ip_input()` function appeared *three times* in the first stack trace, and `ip_output()` appeared twice in the second. These stacks show how the kernel prepared to send single packets.

This was traced on a system with IPSec tunneling and Network Address Translation (NAT) configured. Features such as these can resubmit IP packets back into the network stack for reprocessing. For IPSec, it means that both physical network interface and virtual tunnel interface packets may be traced.

Summary

This chapter showed many ways to observe network I/O details using DTrace from within different layers of the TCP/IP stack. Before DTrace, much of this was typically performed by capturing every packet on the wire (network interface promiscuous mode) and passing it to user-land software for analysis. The scripts in this chapter demonstrated tracing only the events of interest and summarizing information in-kernel before handing to user-land processes, minimizing performance overhead. We also demonstrated the ability to show context information from the system that isn't present in the transmitted packet, such as the process ID for the connection. And, we also demonstrated tracing other stack events, such as network interfaces changing negotiated state, which may not generate packets at all.

This page intentionally left blank

7

Application-Level Protocols

This chapter is a continuation of Chapter 6, Network Lower-Level Protocols, and covers several common application-level network protocols, including HTTP and Network File System (NFS). Using DTrace, you can answer questions about application protocols such as the following.

What NFS clients are performing the most I/O?

What files are NFS clients performing I/O *to*?

What is the latency for HTTP requests?

These can be answered with DTrace. As an example, `nfsv3rwsnoop.d` is a DTrace-based tool to trace NFSv3 reads and writes on the NFS server, showing the client and I/O details:

```
server# nfsv3rwsnoop.d
TIME(us)         CLIENT             OP OFFSET(KB)  BYTES PATHNAME
687663304921     192.168.1.109       R 0            4096 /export/fs1/2g-a-128k
687663305729     192.168.1.109       R 4           28672 /export/fs1/2g-a-128k
687663308909     192.168.1.109       R 32          32768 /export/fs1/2g-a-128k
687663309083     192.168.1.109       R 64          32768 /export/fs1/2g-a-128k
687663309185     192.168.1.109       R 96          32768 /export/fs1/2g-a-128k
687663309240     192.168.1.109       R 128         32768 /export/fs1/2g-a-128k
687663309274     192.168.1.109       R 160         32768 /export/fs1/2g-a-128k
687663315282     192.168.1.109       R 192         32768 /export/fs1/2g-a-128k
```

Although network sniffing tools can examine similar protocol data, they can examine only the information in the protocol headers and provide limited output formats. DTrace can access this information alongside events from anywhere in the operating system stack, showing not just what packets were sent but also why they were sent. DTrace also allows this data to be filtered and summarized in-kernel, resulting in less overhead than would be incurred when network-sniffing tools capture and postprocess every packet.

The "Strategy" and "Checklist" sections for application protocol I/O are similar to those shown in the previous chapter for network stack I/O, with these key differences:

- Application protocols may be processed by user-land daemons, whereas the network stack is typically kernel-only.
- The difference between tracing server- or client-side is more evident for application protocols, because it may involve tracing completely different bodies of software.

Many common application-level network protocols are covered in this chapter; however, there are far more than we can cover here. It should be possible to use DTrace to examine all network protocols implemented by software, since DTrace can examine the operation of all software. For any given application protocol, see the "Strategy" and "Checklist" sections that follow for tips to get you started. The scripts included here for other protocols may also provide useful ideas that can be applied in other ways.

Capabilities

See Figures 6.1 and 6.2 from Chapter 6.

Strategy

To get started using DTrace to examine application protocol I/O, follow these steps (the target of each step is in bold):

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.
2. In addition to those DTrace tools, familiarize yourself with **existing network statistic tools**. For example, you can use `nfsstat` for NFS statistics, and you can use `tcpdump` or `snoop` for packet details including the application protocol. The metrics that these print can be treated as starting points for customization with DTrace.

3. Locate or write tools to generate **known network I/O**, which could be as simple as using `ftp` to transfer a large file of a known size. When testing over NFS and other network shares, regular file system benchmark tools including Filebench can be applied to a mounted share to generate network I/O. It is extremely helpful to have known workloads to examine while developing DTrace scripts.
4. Check which **stable providers** exist and are available on your operating system to examine the protocol, such as the `nfsv3` provider for examining NFSv3. You can use these to write stable one-liners and scripts that should continue to work for future operating system updates.
5. If no stable provider is available, first check whether the protocol is kernel-based (for example, most NFS server and client drivers) or user-land-based (for example, the iSCSI daemon `iscsitgtd`). For kernel-based protocols, check what probes are available in the **sdt** and **fbt providers**; for user-land-based protocols, check the **pid** and **syscall** providers. Simple ways to check include listing the probes and using `grep` and frequency counting events with a known workload.
6. If the **source code** is available, it can be examined to find suitable probe points for either the fbt or pid provider and to see what arguments may be available for these probes. If the source code isn't available, program flow may be determined by tracing entry and return probes with the `flowindent` pragma and also by examining **stack backtraces**.

Checklist

Consider Table 7-1 to be a checklist of application protocol issue types that can be examined using DTrace.

Table 7-1 Network I/O Checklist

Issue	Description
Volume	A server may be accepting a high volume of network I/O from unexpected clients, which may be avoidable by reconfiguring the client environment. Applications may also be performing a high volume of network I/O that could be avoided by modifying their behavior. DTrace can be used to examine network I/O by client, port, and application stack trace to identify who is using the network, how much, and why.

continues

Table 7-1 Network I/O Checklist (*Continued*)

Issue	Description
Latency	There are different latencies to examine for application protocol I/O: • **Operation latency, client side**: The time from an operation request to its completion includes the network latency and target server latency. • **Operation latency, server side**: The time from receiving an operation request to sending its completion represents the latency, which the server is responsible for. If the latency as measured on the client is much higher than that measured on the server, then the missing latency may be from the network, especially if the client is several routing hops away from the server.
Queueing	If the application protocol implements a queue for outstanding operations, DTrace can be used to examine details of the queue. This may include the average queue length and the latency while waiting on the queue. Queueing can have a significant effect on performance.
Errors	Various errors could be encountered and conveyed by application protocols, but the end user may not necessarily be informed that they have occurred. DTrace can check whether errors are occurring and provide details including stack backtraces to understand their nature.
Configuration	If an application protocol can be configured to behave in different ways, such as enabling performance-enhancing features, DTrace can be used to confirm that the intended configuration is taking effect.

Providers

Table 7-2 shows providers of interest when tracing application protocol network I/O.

Table 7-2 Providers for Network I/O

Provider	Description
nfsv3, nfsv4	Stable providers for tracing the NFSv3 and NFSv4 protocols on the server. Details include client information, filenames, and I/O sizes.
smb	Stable provider for tracing the CIFS protocol on the server.
iscsi	Stable provider for tracing the iSCSI protocol on the target server.
fc	Stable provider for tracing Fibre Channel on the target server.
http	Stable USDT provider for tracing HTTP, implemented as a mod_dtrace plug-in for the Apache Web server.
ftp	Stable provider for tracing the FTP protocol, USDT-based.

Table 7-2 Providers for Network I/O (*Continued*)

Provider	Description
syscall	Trace entry and return of operating system calls, arguments, and return values. Network I/O usually begins as application syscalls, making this a useful provider to consider. It also fires in application context where the user stack trace can be examined.
sdt	Kernel-based application protocols may have sdt probes of interest.
fbt	Any kernel-based application protocol can be examined using the fbt provider. As this traces kernel functions, the interface is considered unstable and may change between releases of the operating system and drivers, meaning that scripts based on fbt may need to be slightly rewritten for each such update.
pid	Any user-land-based application protocol can be examined using the pid provider. As this traces functions in the user-land software, the interface is considered unstable and may change between versions of the protocol software.

Check your operating system version to see which of these providers are available. Protocol providers are described in the "Scripts" section along with the protocol scripts; the fbt and pid providers are introduced in the following sections as they apply to multiple protocols.

fbt Provider

The fbt provider can be used to examine *all* the functions for kernel-based protocols, the function arguments, the return codes, the return instruction offsets, and both the elapsed time and the on-CPU time. See the "fbt Provider" chapter of the DTrace Guide for the full reference,[1] and see the "fbt Provider" section in Chapter 12, Kernel.

To navigate the available probes, begin by listing them and search for the protocol name. Here's an example for NFS:

```
solaris# dtrace -ln fbt::: | grep nfs
 4137       fbt             nfs                 nfs3_attr_cache entry
 4138       fbt             nfs                 nfs3_attr_cache return
 4139       fbt             nfs               nfs_getattr_cache entry
 4140       fbt             nfs               nfs_getattr_cache return
 4141       fbt             nfs                 free_async_args entry
```

continues

1. *http://wikis.sun.com/display/DTrace/fbt+Provider*

```
  4142        fbt              nfs                    free_async_args return
  4143        fbt              nfs                    nfs_async_start entry
  4144        fbt              nfs                        nfs_mi_init entry
  4145        fbt              nfs                        nfs_mi_init return
...
```

The function names may match the protocol operations, simply because it can be easy to design the code that way. Another way to quickly navigate fbt probes is to perform a known workload and to frequency count fired probes.

Using the fbt provider should be considered a last resort as it is tracing the source code implementation, which is considered an unstable interface. Check for the availability of stable providers first, for example, the nfsv3 and iscsi providers.

pid Provider

The pid provider can be used to examine all the functions for user-land based protocols, the function arguments, the return codes, the return instruction offsets, and both the elapsed time and the on-CPU time. See the "pid Provider" chapter of the DTrace Guide for the full reference,[2] and see the "pid Provider" section in Chapter 9, Applications.

To navigate the available probes, begin by listing them and search for the protocol name. Here's an example for the RIP protocol implemented by the `in.routed` daemon on Solaris:

```
solaris# dtrace -ln 'pid$target:::entry' -p `pgrep in.routed` | grep -i rip
 7572     pid927         in.routed                            rip_bcast entry
 7573     pid927         in.routed                            rip_query entry
 7666     pid927         in.routed                         rip_strerror entry
 7700     pid927         in.routed                            trace_rip entry
85717     pid927         libc.so.1                         strip_quotes entry
98532     pid927         in.routed                       ripv1_mask_net entry
98533     pid927         in.routed                      ripv1_mask_host entry
98549     pid927         in.routed                             read_rip entry
98560     pid927         in.routed                        open_rip_sock entry
98561     pid927         in.routed                              rip_off entry
98562     pid927         in.routed                          rip_mcast_on entry
98563     pid927         in.routed                         rip_mcast_off entry
98564     pid927         in.routed                               rip_on entry
...
```

This has discovered promising function names such as `rip_bcast()` and `rip_query()`, which can be traced using DTrace. If available, the source code can be examined to see what the arguments to these functions are.

2. *http://wikis.sun.com/display/DTrace/pid+Provider*

As with the fbt provider, the pid provider has the capability to trace the unstable source code implementation and should be considered a last resort if stable providers are not available.

One-Liners

The following one-liners are grouped by provider. Not all providers are available on all operating system versions, especially newer providers, such as nfsv3, nfsv4, smb, iscsi, and fc. See the "Scripts" section for each provider for more details on provider availability.

syscall Provider

HTTP files opened by the httpd server:

```
dtrace -n 'syscall::open*:entry /execname == "httpd"/ { @[copyinstr(arg0)] = count(); }'
```

SSH logins by UID and home directory:

```
dtrace -n 'syscall::chdir:entry /execname == "sshd"/ { printf("UID:%d %s", uid,
copyinstr(arg0)); }'
```

nfsv3 Provider

NFSv3 frequency of NFS operations by type:

```
dtrace -n 'nfsv3::: { @[probename] = count(); }'
```

NFSv3 count of operations by client address:

```
dtrace -n 'nfsv3:::op-*-start { @[args[0]->ci_remote] = count(); }'
```

NFSv3 count of operations by file path name:

```
dtrace -n 'nfsv3:::op-*-start { @[args[1]->noi_curpath] = count(); }'
```

NFSv3 total read payload bytes, requested:

```
dtrace -n 'nfsv3:::op-read-start { @ = sum(args[2]->count); }'
```

NFSv3 total read payload bytes, completed:

```
dtrace -n 'nfsv3:::op-read-done { @ = sum(args[2]->res_u.ok.data.data_len); }'
```

NFSv3 read I/O size distribution:

```
dtrace -n 'nfsv3:::op-read-start { @ = quantize(args[2]->count); }'
```

NFSv3 total write payload bytes, requested:

```
dtrace -n 'nfsv3:::op-write-start { @ = sum(args[2]->data.data_len); }'
```

NFSv3 total write payload bytes, completed:

```
dtrace -n 'nfsv3:::op-write-done { @ = sum(args[2]->res_u.ok.count); }'
```

NFSv3 write I/O size distribution:

```
dtrace -n 'nfsv3:::op-write-start { @ = quantize(args[2]->data.data_len); }'
```

NFSv3 error frequency by type:

```
dtrace -n 'nfsv3:::op-*-done { @[probename, args[2]->status] = count(); }'
```

nfsv4 Provider

NFSv4 frequency of NFS operations and compound operations by type:

```
dtrace -n 'nfsv4::: { @[probename] = count(); }'
```

NFSv4 count of operations by client address:

```
dtrace -n 'nfsv4:::op-*-start { @[args[0]->ci_remote] = count(); }'
```

NFSv4 count of operations by file path name:

```
dtrace -n 'nfsv4:::op-*-start { @[args[1]->noi_curpath] = count(); }'
```

NFSv4 total read payload bytes, requested:

```
dtrace -n 'nfsv4:::op-read-start { @ = sum(args[2]->count); }'
```

NFSv4 total read payload bytes, completed:

```
dtrace -n 'nfsv4:::op-read-done { @ = sum(args[2]->data_len); }'
```

NFSv4 read I/O size distribution:

```
dtrace -n 'nfsv4:::op-read-start { @ = quantize(args[2]->count); }'
```

NFSv4 total write payload bytes, requested:

```
dtrace -n 'nfsv4:::op-write-start { @ = sum(args[2]->data_len); }'
```

NFSv4 total write payload bytes, completed:

```
dtrace -n 'nfsv4:::op-write-done { @ = sum(args[2]->count); }'
```

NFSv4 write I/O size distribution:

```
dtrace -n 'nfsv4:::op-write-start { @ = quantize(args[2]->data_len); }'
```

NFSv4 error frequency by type:

```
dtrace -n 'nfsv4:::op-*-done { @[probename, args[2]->status] = count(); }'
```

smb Provider

CIFS frequency of operations by type:

```
dtrace -n 'smb::: { @[probename] = count(); }'
```

CIFS count of operations by client address:

```
dtrace -n 'smb:::op-*-start { @[args[0]->ci_remote] = count(); }'
```

CIFS count of operations by file path name:

```
dtrace -n 'smb:::op-*-done { @[args[1]->soi_curpath] = count(); }'
```

CIFS total read payload bytes:

```
dtrace -n 'smb:::op-Read*-start { @ = sum(args[2]->soa_count); }'
```

CIFS read I/O size distribution:

```
dtrace -n 'smb:::op-Read*-start { @ = quantize(args[2]->soa_count); }'
```

CIFS total write payload bytes:

```
dtrace -n 'smb:::op-Write*-start { @ = sum(args[2]->soa_count); }'
```

CIFS write I/O size distribution:

```
dtrace -n 'smb:::op-Write*-start { @ = quantize(args[2]->soa_count); }'
```

http Provider

HTTP frequency count requested URIs:

```
dtrace -n 'http*:::request-start { @[args[1]->hri_uri] = count(); }'
```

HTTP frequency count response codes:

```
dtrace -n 'http*:::request-done { @[args[1]->hri_respcode] = count(); }'
```

HTTP summarize user agents:

```
dtrace -n 'http*:::request-start { @[args[1]->hri_useragent] = count(); }'
```

iscsi Provider

iSCSI command type frequency:

```
dtrace -n 'iscsi*::: { @[probename] = count(); }'
```

iSCSI count of operations by client address:

```
dtrace -n 'iscsi*::: { @[args[0]->ci_remote] = count(); }'
```

iSCSI payload bytes by operation type:

```
dtrace -n 'iscsi*::: { @[probename] = sum(args[1]->ii_datalen); }'
```

iSCSI payload size distribution by operation type:

```
dtrace -n 'iscsi*::: { @[probename] = quantize(args[1]->ii_datalen); }'
```

fc Provider

FC command type frequency:

```
dtrace -n 'fc::: { @[probename] = count(); }'
```

FC count of operations by client address:

```
dtrace -n 'fc::: { @[args[0]->ci_remote] = count(); }'
```

FC bytes transferred:

```
dtrace -n 'fc:::xfer-start { @ = sum(args[4]->fcx_len); }'
```

FC transfer size distribution:

```
dtrace -n 'fc:::xfer-start { @ = quantize(args[4]->fcx_len); }'
```

The following sections demonstrate selected one-liners from these categories.

syscall Provider Examples

HTTP Files Opened by the httpd Server

Files opened by `httpd` processes are typically those that were requested by HTTP clients, so frequency counting these gives a sense of what is being served by the HTTP server:

```
server# dtrace -n 'syscall::open*:entry /execname == "httpd"/ { @[copyinstr(arg0)] =
count(); }'
dtrace: description 'syscall::open*:entry ' matched 4 probes
dtrace: error on enabled probe ID 3 (ID 14361: syscall::openat:entry): invalid
address
 (0xffd19652) in action #2 at DIF offset 28
[...output truncated...]
  /usr/lib/ak/htdocs/wiki/index.php                                 10
  /usr/lib/ak/htdocs/wiki/languages/DynamicPageList2.i18n.php       10
  /usr/lib/ak/htdocs/wiki/languages/DynamicPageList2Include.php     10
  /usr/lib/ak/htdocs/wiki/languages/Language.php                    10
  /usr/lib/ak/htdocs/wiki/languages/LoopFunctions.i18n.php          10
  /usr/lib/ak/htdocs/wiki/languages/Names.php                       10
  /usr/lib/ak/htdocs/wiki/languages/messages/MessagesEn.php         10
  /var/php/5.2/pear/DynamicPageList2.i18n.php                       10
  /var/php/5.2/pear/DynamicPageList2Include.php                     10
```

```
   /var/php/5.2/pear/LoopFunctions.i18n.php                              10
   /var/php/5.2/sessions/sess_t2v0ioigsrupgrap4lib43vve3                 10
   /usr/lib/ak/htdocs/wiki/includes/SkinTemplate.php                     12
   /usr/lib/ak/htdocs/wiki/DynamicPageList2.i18n.php                     20
   /usr/lib/ak/htdocs/wiki/DynamicPageList2Include.php                   20
   /usr/lib/ak/htdocs/wiki/LoopFunctions.i18n.php                        20
   /.htaccess                                                            63
   /usr/.htaccess                                                        63
   /usr/lib/.htaccess                                                    63
   /usr/lib/ak/.htaccess                                                 63
```

While tracing, a wiki page was loaded from the HTTP server. The previous (truncated) output shows the various files that were read to serve this request and the counts. The output also contains an error as the `open*` probe definition matched `openat()` by accident, which does not contain a string as the first argument; this could be improved by rewriting as a script and listing the desired variants of `open()` only.

SSH Logins by UID and Home Directory

This one-liner traces successful SSH logins showing the UID and home directory. It works by relying on how the SSH server daemon (`sshd`) processes a login: Both the current Solaris and Mac OS X versions execute `chdir()` to the home directory after setting the UID to the logged-in user, which is traced:

```
server# dtrace -n 'syscall::chdir:entry /execname == "sshd"/ { printf("UID:%d %s",
uid, copyinstr(arg0)); }'
dtrace: description 'syscall::chdir:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  9  14265                      chdir:entry UID:130948 /home/brendan
```

This captured a login by UID 130948, with the home directory `/home/brendan`.

NFSv3 Provider Examples

NFSv3 Frequency of NFS Operations by Type

Frequency counting NFSv3 operation types gives an idea of the current NFSv3 workload:

```
server# dtrace -n nfsv3::: { @[probename] = count(); }'
dtrace: description 'nfsv3::: ' matched 44 probes
^C

  op-lookup-done                                                         1
  op-lookup-start                                                        1
  op-readdirplus-done                                                    1
```

continues

```
    op-readdirplus-start                                           1
    op-setattr-done                                                1
    op-setattr-start                                               1
    op-access-done                                                 2
    op-access-start                                                2
    op-write-done                                                  2
    op-write-start                                                 2
    op-getattr-done                                                6
    op-getattr-start                                               6
    op-read-done                                                1961
    op-read-start                                               1961
```

As this one-liner executed, there were 1,961 NFSv3 reads and 2 NFSv3 writes, along with some other operation types. This one-liner traces both the `start` and `done` events for each operation.

NFSv3 Count of Operations by Client Address

This is a quick way to determine which clients are using an NFSv3 server and how many operations there are:

```
server# dtrace -n nfsv3:::op-*-start { @[args[0]->ci_remote] = count(); }'
dtrace: description 'nfsv3:::op-*-start ' matched 22 probes
^C

  192.168.2.40                                                    42
  192.168.2.30                                                  2888
```

The host 192.168.2.30 performed 2888 NFSv3 operations while the one-liner was tracing.

NFSv3 Count of Operations by File Path Name

The filename for all NFSv3 operations can be easily traced from the provider arguments:

```
server# dtrace -n 'nfsv3:::op-*-start { @[args[1]->noi_curpath] = count(); }'
dtrace: description 'nfsv3:::op-*-start ' matched 22 probes
^C

  /export/fs2                                                      1
  /export/fs1                                                      2
  /export/fs1/1k 5
  /export/fs2/100g                                                42
  /export/fs1/100g                                              1131
```

The hottest file was `/export/fs1/100g`, which had 1,131 NFSv3 operations while tracing.

NFSv3 Read I/O Size Distribution

The size of NFS I/O can have a significant impact on performance, especially unusually large or small I/O sizes. The distribution can be examined with the DTrace `quantize()` function:

```
server# dtrace -n 'nfsv3:::op-read-start { @ = quantize(args[2]->count); }'
dtrace: description 'nfsv3:::op-read-start ' matched 1 probe
^C


           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |                                         54
            1024 |@@                                       414
            2048 |@@@@@@@@@@@@@                            2873
            4096 |@@@@@@@@@@@@@@@@@@@@@                    4713
            8192 |@@@@                                     1012
           16384 |                                         0
```

This shows that most of the NFSv3 reads were between 2KB and 8KB in size.

NFSv4 Provider Examples

Most of the NFSv4 one-liners produce output similar to that demonstrated for NFSv3.

NFSv4 Frequency of NFS Operations and Compound Operations by Type

Frequency counting NFSv4 provider event types gives an idea of the current NFSv4 workload:

```
server# dtrace -n 'nfsv4::: { @[probename] = count(); }'
dtrace: description 'nfsv4::: ' matched 81 probes
^C

  op-access-done                                                    1
  op-access-start                                                   1
  op-commit-done                                                    1
  op-commit-start                                                   1
  op-lookup-done                                                    1
  op-lookup-start                                                   1
  op-nverify-done                                                   1
  op-nverify-start                                                  1
  op-write-done                                                     1
  op-write-start                                                    1
  op-close-done                                                     2
  op-close-start                                                    2
  op-open-done                                                      2
  op-open-start                                                     2
  op-restorefh-done                                                 2
  op-restorefh-start                                                2
  op-savefh-done                                                    2
```
continues

```
  op-savefh-start                                                2
  op-getfh-done                                                  3
  op-getfh-start                                                 3
  op-getattr-done                                                8
  op-getattr-start                                               8
  op-read-done                                                 115
  op-read-start                                                115
  compound-done                                                125
  compound-start                                               125
  op-putfh-done                                                125
  op-putfh-start                                               125
```

Unlike the NFSv3 example, this now contains counts for compound operations: `compound-start` and `compound-done`. This particular example shows the ratio between normal to compound operations to be about 2x.

smb Provider Examples

CIFS Frequency of Operations by Type

Frequency counting CIFS operations gives an idea of the current CIFS workload:

```
server# dtrace -n 'smb::: { @[probename] = count(); }'
dtrace: description 'smb::: ' matched 120 probes
^C

  op-Close-done                                                  3
  op-Close-start                                                 3
  op-NtCreateX-done                                              3
  op-NtCreateX-start                                             3
  op-WriteX-done                                                16
  op-WriteX-start                                               16
  op-Transaction2-done                                         158
  op-Transaction2-start                                        158
  op-ReadX-done                                                803
  op-ReadX-start                                               803
```

While this one-liner was running, there were 803 ReadX operations.

CIFS Count of Operations by Client Address

This is a quick way to determine which clients are using a CIFS server and how many operations there are:

```
server# dtrace -n 'smb:::op-*-start { @[args[0]->ci_remote] = count(); }'
dtrace: description 'smb:::op-*-start ' matched 60 probes
^C

  192.168.3.103                                                867
  192.168.3.102                                               2162
```

The client 192.168.3.102 performed 2,162 CIFS operations while the one-liner was tracing.

CIFS Count of Operations by File Path Name

While this one-liner was tracing, the `/export/fs1/100g` file had 1,163 CIFS operations. The filename is printed only if available for that operation type and known; otherwise, it is listed as `<unknown>`, which was the case for eight operations. Further DTracing can examine them in more detail if desired.

```
server# dtrace -n 'smb:::op-*-done { @[args[1]->soi_curpath] = count(); }'
dtrace: description 'smb:::op-*-done ' matched 60 probes
^C

  <unknown>                                                         8
  /export/fs1/8k                                                  164
  /export/fs1/100g                                               1163
```

CIFS Read I/O Size Distribution

This shows that most of the CIFS read I/O while tracing was between 4KB and 8KB in size.

```
server# dtrace -n 'smb:::op-Read*-start { @ = quantize(args[2]->soa_count); }'
dtrace: description 'smb:::op-Read*-start ' matched 3 probes
^C


           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@                                      98
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    1031
            8192 |                                         0
```

http Provider Examples

HTTP Summarize User Agents

This is a quick way to see what software HTTP clients are using to browse your Web server. The most popular browser while this one-liner was tracing was Mozilla/Firefox.

```
server# dtrace -n 'http*:::request-start { @[args[1]->hri_useragent] = count(); }'
dtrace: description 'http*:::request-start ' matched 10 probes
^C

  Lynx/2.8.5rel.1 libwww-FM/2.14                                   2
  ELinks/0.11.6 (textmode; SunOS 5.11 i86pc; 96x41-3)              11
```

continues

```
 Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.1.9) Gecko/20100315
Firefox/3.5.9                43
```

More examples of http provider one-liners are in the "http Scripts" section.

Scripts

Table 7-3 summarizes the scripts that follow and the providers they use. For network file system protocols such as NFS, Chapter 5, File Systems, contains additional scripts for client-side tracing.

Table 7-3 Network Script Summary

Script	Protocol	Description	Provider
`nfsv3rwsnoop.d`	NFSv3	NFSv3 read/write snoop, showing client, path name, and bytes	nfsv3
`nfsv3ops.d`	NFSv3	Shows who is calling what NFSv3 operations	nfsv3
`nfsv3fileio.d`	NFSv3	Shows NFSv3 read and write bytes by filename	nfsv3
`nfsv3rwtime.d`	NFSv3	Measures NFSv3 read and write latency	nfsv3
`nfsv3syncwrite.d`	NFSv3	Identifies synchronous NFSv3 writes and commits	nfsv3
`nfsv3commit.d`	NFSv3	Shows NFSv3 commit operation details	nfsv3
`nfsv3errors.d`	NFSv3	Traces NFSv3 errors live	nfsv3
`nfsv3fbtrws.d`	NFSv3	fbt provider version of `nfsv3rwsnoop.d`	fbt
`nfsv3disk.d`	NFSv3	Reads/writes throughput at the NFS, ZFS, and disk layers	nfsv3, io, sdt
`nfsv4rwsnoop.d`	NFSv4	NFSv4 read/write snoop, showing client, pathname, and bytes	nfsv4
`nfsv4ops.d`	NFSv4	Shows who is calling what NFSv4 operations	nfsv4
`nfsv4fileio.d`	NFSv4	Shows NFSv4 read and write bytes by filename	nfsv4
`nfsv4rwtime.d`	NFSv4	Measures NFSv4 read and write latency	nfsv4
`nfsv4syncwrite.d`	NFSv4	Identifies synchronous NFSv4 writes and commits	nfsv4
`nfsv4commit.d`	NFSv4	Shows NFSv4 commit operation details	nfsv4

Table 7-3 Network Script Summary (*Continued*)

Script	Protocol	Description	Provider
`nfsv4errors.d`	NFSv4	Traces NFSv4 errors live	nfsv4
`nfsv4deleg.d`	NFSv4	Trace NFSv4 write delegation events	fbt
`cifsrwsnoop.d`	CIFS	CIFS read/write snoop, showing client, path name, and bytes	smb
`cifsops.d`	CIFS	Shows who is calling what CIFS operations	smb
`cifsfileio.d`	CIFS	Shows CIFS read and write bytes by filename	smb
`cifsrwtime.d`	CIFS	Measures CIFS read and write latency	smb
`cifserrors.d`	CIFS	Traces CIFS errors live	smb
`cifsfbtnofile.d`	CIFS	Traces CIFS no such file errors with path name and share	fbt
`httpclients.d`	HTTP	Summarizes HTTP client throughput	http
`httperrors.d`	HTTP	Summarizes HTTP errors	http
`httpio.d`	HTTP	Shows HTTP send/receive size distribution	http
`httpdurls.d`	HTTP	Counts HTTP `GET` requests by URL	syscall
`weblatency.d`	HTTP	Shows client HTTP `GET`s by Web server and latency	syscall
`getaddrinfo.d`	DNS	Show latency of client getaddrinfo() lookups	pid
`dnsgetname.d`	DNS	Traces DNS queries on a BIND server	pid
`ftpdxfer.d`	FTP	Traces FTP data transfers with client, path, and other details	ftp
`ftpdfileio.d`	FTP	Summarizes FTP data bytes by filename	ftp
`proftpdcmd.d`	FTP	Traces proftpd FTP commands	pid
`tnftpdcmd.d`	FTP	Traces tnftpd FTP commands	pid
`proftpdtime.d`	FTP	Shows FTP command latency	pid
`proftpdio.d`	FTP	FTP server iostat, for FTP operations	pid
`iscsiwho.d`	iSCSI	Shows iSCSI clients and probe counts from the target server	iscsi
`iscsirwsnoop.d`	iSCSI	Traces iSCSI events on the target server	iscsi
`iscsirwtime.d`	iSCSI	Measures iSCSI read/write latency from the target server	iscsi
`iscsicmds.d`	iSCSI	Show iSCSI commands by SCSI command type	iscsi

continues

Table 7-3 Network Script Summary (*Continued*)

Script	Protocol	Description	Provider
`iscsiterr.d`	iSCSI	Trace iSCSI errors on the target server	fbt
`fcwho.d`	FC	Shows FC clients and probe counts from the target server	fc
`fcerror.d`	FC	Traces FC errors with various details	fbt
`sshcipher.d`	SSH	Measures SSH client encryption/compression overhead	pid
`sshdactivity.d`	SSH	Identifies active SSH activity service side	syscall
`sshconnect.d`	SSH	Identifies SSH client connect latency	syscall
`scpwatcher.d`	SSH	Monitor scp progress systemwide	syscall
`nismatch.d`	NIS	Traces NIS map match requests on the NIS server	pid
`ldapsyslog.d`	LDAP	Traces OpenLDAP requests on the LDAP server	pid

The fbt, sdt, and pid providers are considered "unstable" interfaces, because they instrument a specific operating system or application version. For this reason, scripts that use these providers may require changes to match the version of the software you are using. These scripts have been included here as examples of D programming and of the kind of data that DTrace can provide for each of these topics. See Chapter 12 for more discussion about using the fbt provider.

NFSv3 Scripts

NFS is the Network File System protocol for sharing files over the network using a file system interface. The scripts in this section are for tracing NFS version 3 (NFSv3) events on an NFS server. For NFSv3 client-side tracing, see Chapter 5.

Most of these scripts use the nfsv3 provider, which is fully documented in the nfsv3 provider section of the DTrace Guide.[3] It is currently available in OpenSolaris[4] and Solaris Nevada.[5] Listing the nfsv3 probes on Solaris Nevada, circa June 2010, yields the following:

3. *http://wikis.sun.com/display/DTrace/nfsv3+Provider*
4. PSARC 2008/050, CR 6660173, was integrated into Solaris Nevada in February 2008 (snv_84).
5. It is also shipped as part of the Oracle Sun ZFS Storage Appliance, where it powers NFSv3 Analytics.

```
solaris# dtrace -ln nfsv3:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
11363      nfsv3           nfssrv                       rfs3_commit op-commit-done
11365      nfsv3           nfssrv                       rfs3_commit op-commit-start
11366      nfsv3           nfssrv                     rfs3_pathconf op-pathconf-done
11368      nfsv3           nfssrv                     rfs3_pathconf op-pathconf-start
11369      nfsv3           nfssrv                       rfs3_fsinfo op-fsinfo-done
11371      nfsv3           nfssrv                       rfs3_fsinfo op-fsinfo-start
11372      nfsv3           nfssrv                       rfs3_fsstat op-fsstat-done
11374      nfsv3           nfssrv                       rfs3_fsstat op-fsstat-start
11375      nfsv3           nfssrv                  rfs3_readdirplus op-readdirplus-done
11377      nfsv3           nfssrv                  rfs3_readdirplus op-readdirplus-start
11378      nfsv3           nfssrv                      rfs3_readdir op-readdir-done
11380      nfsv3           nfssrv                      rfs3_readdir op-readdir-start
11381      nfsv3           nfssrv                         rfs3_link op-link-done
11384      nfsv3           nfssrv                         rfs3_link op-link-start
11385      nfsv3           nfssrv                       rfs3_rename op-rename-done
11387      nfsv3           nfssrv                       rfs3_rename op-rename-start
11388      nfsv3           nfssrv                        rfs3_rmdir op-rmdir-done
11390      nfsv3           nfssrv                        rfs3_rmdir op-rmdir-start
11391      nfsv3           nfssrv                       rfs3_remove op-remove-done
11393      nfsv3           nfssrv                       rfs3_remove op-remove-start
11394      nfsv3           nfssrv                        rfs3_mknod op-mknod-done
11396      nfsv3           nfssrv                        rfs3_mknod op-mknod-start
11397      nfsv3           nfssrv                      rfs3_symlink op-symlink-done
11399      nfsv3           nfssrv                      rfs3_symlink op-symlink-start
11400      nfsv3           nfssrv                        rfs3_mkdir op-mkdir-done
11402      nfsv3           nfssrv                        rfs3_mkdir op-mkdir-start
11403      nfsv3           nfssrv                       rfs3_create op-create-done
78882      nfsv3           nfssrv                       rfs3_create op-create-start
78883      nfsv3           nfssrv                        rfs3_write op-write-done
78885      nfsv3           nfssrv                        rfs3_write op-write-start
78888      nfsv3           nfssrv                         rfs3_read op-read-done
78890      nfsv3           nfssrv                         rfs3_read op-read-start
78891      nfsv3           nfssrv                     rfs3_readlink op-readlink-done
78894      nfsv3           nfssrv                     rfs3_readlink op-readlink-start
78895      nfsv3           nfssrv                       rfs3_access op-access-done
78897      nfsv3           nfssrv                       rfs3_access op-access-start
78898      nfsv3           nfssrv                       rfs3_lookup op-lookup-done
78900      nfsv3           nfssrv                       rfs3_lookup op-lookup-start
78901      nfsv3           nfssrv                      rfs3_setattr op-setattr-done
78903      nfsv3           nfssrv                      rfs3_setattr op-setattr-start
78904      nfsv3           nfssrv                      rfs3_getattr op-getattr-done
78905      nfsv3           nfssrv                      rfs3_getattr op-getattr-start
78940      nfsv3           nfssrv                       rpc_null_v3 op-null-done
78941      nfsv3           nfssrv                       rpc_null_v3 op-null-start
```

NFSv3 operations can be traced with the `op-*-start` and `op-*-done` probes. Each provides arguments for the operation, including client address and filename (when appropriate). The previous listing also highlights the locations of the probes in the nfssrv kernel module by showing the kernel functions that contain them (`FUNCTION` column). These can be treated as starting points if you need to examine the source code.

If the nfsv3 provider is not available, the fbt provider can be used instead, bearing in mind that fbt-based scripts may only execute on the kernel version they were written for. Finding the right kernel functions to trace using the fbt provider can sometimes be a challenge. However, if the kernel version you are using is anything like the Solaris Nevada version shown earlier, the function names may be similar to the operation names, making them easy to find. An example of fbt provider tracing of NFSv3 is included in this section: `nfsv3fbtrws.d`.

nfsv3rwsnoop.d

This script traces NFSv3 read and write requests, printing a line of output for each operation as they occur.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-10s %6s %s\n", "TIME(us)",
            "CLIENT", "OP", "OFFSET(KB)", "BYTES", "PATHNAME");
}

nfsv3:::op-read-start
{
        printf("%-16d %-18s %2s %-10d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "R", args[2]->offset / 1024,
            args[2]->count, args[1]->noi_curpath);
}

nfsv3:::op-write-start
{
        printf("%-16d %-18s %2s %-10d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, "W", args[2]->offset / 1024,
            args[2]->data.data_len, args[1]->noi_curpath);
}
```

Script nfsv3rwsnoop.d

Examples

To become familiar with this script, different workloads are traced.

Streaming Read. Here a large file was read sequentially over NFSv3, creating a streaming read workload:

```
server# nfsv3rwsnoop.d
TIME(us)         CLIENT             OP OFFSET(KB)  BYTES PATHNAME
687663304921     192.168.1.109       R 0            4096 /export/fs1/2g-a-128k
687663305729     192.168.1.109       R 4           28672 /export/fs1/2g-a-128k
687663308909     192.168.1.109       R 32          32768 /export/fs1/2g-a-128k
687663309083     192.168.1.109       R 64          32768 /export/fs1/2g-a-128k
687663309185     192.168.1.109       R 96          32768 /export/fs1/2g-a-128k
687663309240     192.168.1.109       R 128         32768 /export/fs1/2g-a-128k
687663309274     192.168.1.109       R 160         32768 /export/fs1/2g-a-128k
```

```
687663315282     192.168.1.109       R 192          32768 /export/fs1/2g-a-128k
687663318259     192.168.1.109       R 224          32768 /export/fs1/2g-a-128k
687663320669     192.168.1.109       R 256          32768 /export/fs1/2g-a-128k
687663323752     192.168.1.109       R 288          32768 /export/fs1/2g-a-128k
[...]
```

We can see that this is a sequential streaming workload: The offsets are printed in KB, and comparing them with the I/O size in the BYTES column shows that the requested offsets are sequential. The I/O size also quickly increases to 32KB, evidence of a streaming workload.

Random Read

Now the same file is read by a program that performs random reads, with an I/O size of 512 bytes:

```
server# nfsv3rwsnoop.d
TIME(us)         CLIENT             OP OFFSET(KB)  BYTES PATHNAME
687710632217     192.168.1.109       R 1224048      4096 /export/fs1/2g-a-128k
687710632915     192.168.1.109       R 1794396      4096 /export/fs1/2g-a-128k
687710633549     192.168.1.109       R 1164408      4096 /export/fs1/2g-a-128k
687710634181     192.168.1.109       R 723352       4096 /export/fs1/2g-a-128k
687710634855     192.168.1.109       R 135364       4096 /export/fs1/2g-a-128k
687710635516     192.168.1.109       R 1164108      4096 /export/fs1/2g-a-128k
687710636189     192.168.1.109       R 2049512      4096 /export/fs1/2g-a-128k
687710636859     192.168.1.109       R 1406584      4096 /export/fs1/2g-a-128k
687710637522     192.168.1.109       R 142280       4096 /export/fs1/2g-a-128k
687710638260     192.168.1.109       R 1000848      4096 /export/fs1/2g-a-128k
687710638925     192.168.1.109       R 1458220      4096 /export/fs1/2g-a-128k
[...]
```

The offsets look random. Note that there were 4,096 bytes per I/O, despite requesting 512 bytes from the application. This kind of information can be used to tune the network stack to better handle the application workload.

Application Writes. Here the DTraceToolkit was installed into the share. The order that the files were written is clearly visible in the output:

```
server# nfsv3rwsnoop.d
TIME(us)     CLIENT        OP OFFSET(KB) BYTES PATHNAME
688264719673 192.168.1.109  W 0          2716 /export/fs1/DTT/JavaScript/js_objgc.d
688264723221 192.168.1.109  W 0          2373 /export/fs1/DTT/JavaScript/js_flowinfo.d
688264726587 192.168.1.109  W 0          1461 /export/fs1/DTT/JavaScript/js_objnew.d
688264729517 192.168.1.109  W 0          2439 /export/fs1/DTT/JavaScript/Readme
688264736644 192.168.1.109  W 0          3396 /export/fs1/DTT/JavaScript/js_calltime.d
688264739802 192.168.1.109  W 0          2327 /export/fs1/DTT/JavaScript/js_stat.d
688264806739 192.168.1.109  W 0          2602 /export/fs1/DTT/JavaScript/js_cpudist.d
688264809810 192.168.1.109  W 0          1366 /export/fs1/DTT/JavaScript/js_execs.d
688264814143 192.168.1.109  W 0          1458 /export/fs1/DTT/JavaScript/js_who.d
688264817147 192.168.1.109  W 0          1915 /export/fs1/DTT/JavaScript/js_flow.d
[...]
```

nfsv3ops.d

The `nfsv3ops.d` script is presented in this section.

Script

This script shows who is calling what NFSv3 operations. An output summary is printed every five seconds.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing NFSv3 operations... Interval 5 secs.\n");
}

nfsv3:::op-*-start
{
        @ops[args[0]->ci_remote, probename] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n   %-32s %-28s %8s\n", "Client", "Operation", "Count");
        printa("   %-32s %-28s %@8d\n", @ops);
        trunc(@ops);
}
```

Script nfsv3ops.d

Example

This script identifies a read/write workload from the client 192.168.1.109, with writes dominating in the first five-second interval. Each of the NFS operations can be investigated in more detail with DTrace.

```
server# nfsv3ops.d
Tracing NFSv3 operations... Interval 5 secs.

   Client                           Operation                       Count
   192.168.1.109                    op-readlink-start                   2
   192.168.1.109                    op-readdirplus-start                8
   192.168.1.109                    op-access-start                    40
   192.168.1.109                    op-getattr-start                   86
   192.168.1.109                    op-read-start                     934
   192.168.1.109                    op-write-start                   1722

   Client                           Operation                       Count
   192.168.100.3                    op-access-start                     1
   192.168.100.3                    op-readdirplus-start                1
   192.168.110.3                    op-access-start                     1
   192.168.110.3                    op-readdirplus-start                1
```

```
   192.168.100.3                   op-getattr-start                  2
   192.168.110.3                   op-getattr-start                  2
   192.168.1.109                   op-read-start                  1473
   192.168.1.109                   op-write-start                 1477
[...]
```

nfsv3fileio.d

The `nfsv3fileio.d` is a simple script to trace NFSv3 reads and writes, generating a report by filename when Ctrl-C ends tracing.

Script

If the script produces too many lines of output (because of too many different files being accessed), it could be enhanced with `trunc()` to print only the top N read or written files.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... Hit Ctrl-C to end.\n");
}

nfsv3:::op-read-done
{
        @readbytes[args[1]->noi_curpath] = sum(args[2]->res_u.ok.data.data_len);
}

nfsv3:::op-write-done
{
        @writebytes[args[1]->noi_curpath] = sum(args[2]->res_u.ok.count);
}

dtrace:::END
{
        printf("\n%12s %12s  %s\n", "Rbytes", "Wbytes", "Pathname");
        printa("%@12d %@12d  %s\n", @readbytes, @writebytes);
}
```

Script nfsv3fileio.d

Example

While this script was tracing, about 200MB were written to the `/export/fs1/db1` file via NFSv3.

```
server# nfsv3fileio.d
Tracing... Hit Ctrl-C to end.
^C
```

continues

```
    Rbytes        Wbytes  Pathname
         0     206864384  /export/fs1/db1
   1277952             0  /export/fs1/small
  49655808             0  /export/fs1/2g-e-8k
  56111104             0  /export/fs1/2g-e-128k
```

nfsv3rwtime.d

The `nfsv3rwtime.d` script measures NFSv3 read and write operation latency, as observed on the server. This latency includes time querying the underlying file system cache and for disk I/O if required. Latency distribution plots are printed, along with summaries by host and file.

Script

Line 15 saves a time stamp when the I/O starts, which is retrieved during the done probe on line 22 so that the elapsed time for the I/O can be calculated. It is saved in an associative array called `start`, which is keyed on `args[1]->noi_xid`—the transaction identifier for the NFS I/O—so that the done probe is retrieving correct start time stamp for the current I/O.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   inline int TOP_FILES = 10;
6
7   dtrace:::BEGIN
8   {
9           printf("Tracing... Hit Ctrl-C to end.\n");
10  }
11
12  nfsv3:::op-read-start,
13  nfsv3:::op-write-start
14  {
15          start[args[1]->noi_xid] = timestamp;
16  }
17
18  nfsv3:::op-read-done,
19  nfsv3:::op-write-done
20  /start[args[1]->noi_xid] != 0/
21  {
22          this->elapsed = timestamp - start[args[1]->noi_xid];
23          @rw[probename == "op-read-done" ? "read" : "write"] =
24              quantize(this->elapsed / 1000);
25          @host[args[0]->ci_remote] = sum(this->elapsed);
26          @file[args[1]->noi_curpath] = sum(this->elapsed);
27          start[args[1]->noi_xid] = 0;
28  }
29
30  dtrace:::END
31  {
32          printf("NFSv3 read/write distributions (us):\n");
33          printa(@rw);
34
```

```
        printf("\nNFSv3 read/write by host (total us):\n");
        normalize(@host, 1000);
        printa(@host);

        printf("\nNFSv3 read/write top %d files (total us):\n", TOP_FILES);
        normalize(@file, 1000);
        trunc(@file, TOP_FILES);
        printa(@file);
}
```

Script nfsv3rwtime.d

Example

The latency for the reads and writes can be seen in the distribution plots: Reads were usually between 12 and 63 microseconds, as were writes. Such fast times suggest that these reads and writes are returning from cache. DTrace can be used to investigate further.

```
server# nfsv3rwtime.d
Tracing... Hit Ctrl-C to end.
^C
NFSv3 read/write distributions (us):

  read
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@                                       762
              16 |@@@@@@@@@@@@@@@@@@                       7915
              32 |@@@@@@@@@@@@@@@@@                        7117
              64 |@@@                                      1385
             128 |                                         53
             256 |                                         0

  write
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@               14821
              64 |@@@@@@@@@                                5195
             128 |@@@@@                                    2575
             256 |                                         41
             512 |                                         4
            1024 |                                         0
            2048 |                                         1
            4096 |                                         0


NFSv3 read/write by host (total us):

  192.168.1.109                                              2350760

NFSv3 read/write top 10 files (total us):

  /export/fs1/2g-e-8k                                         275704
  /export/fs1/2g-e-128k                                       341566
  /export/fs1/db1                                            1733489
```

nfsv3syncwrite.d

When capacity planning NFS servers, it's important to know whether applications are performing asynchronous or synchronous writes and how much of each. If they are performing synchronous writes, technologies such as flash memory–based separate intent log devices may be added to improve performance. These devices can be expensive, so it's important to know whether they will be needed.

Synchronous writes occur if the NFS write has a stable flag set or if NFS is sending frequent commit operations. The `nfsv3syncwrite.d` script measures these.

Script

To convert from the numeric `stable_how` protocol codes into human-readable strings, a translation table is created on lines 9 to 10 using an associative array.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           /* See /usr/include/nfs/nfs.h */
8           stable_how[0] = "Unstable";
9           stable_how[1] = "Data_Sync";
10          stable_how[2] = "File_Sync";
11          printf("Tracing NFSv3 writes and commits... Hit Ctrl-C to end.\n");
12  }
13
14  nfsv3:::op-write-start
15  {
16          @["write", stable_how[args[2]->stable], args[1]->noi_curpath] = count();
17  }
18
19  nfsv3:::op-commit-start
20  {
21          @["commit", "-", args[1]->noi_curpath] = count();
22  }
23
24  dtrace:::END
25  {
26          printf(" %-7s %-10s %-10s %s\n", "OP", "TYPE", "COUNT", "PATH");
27          printa(" %-7s %-10s %@-10d %s\n", @);
28  }
```

Script nfsv3syncwrite.d

Example

To test this script, writes will be performed to two files: Default asynchronous writes will be performed to `defaultwrite`, and synchronous writes (`O_DSYNC`) will be performed to `syncwrite`. The script shows the following:

```
server# nfsv3syncwrite.d
Tracing NFSv3 writes and commits... Hit Ctrl-C to end.
^C
 OP      TYPE       COUNT      PATH
 commit  -          36         /export/fs1/defaultwrite
 write   File_Sync  22755      /export/fs1/syncwrite
 write   Unstable   32768      /export/fs1/defaultwrite
```

The writes to the `syncwrite` file were all of type `File_Sync`, and to the `defaultwrite` file they were all of type `Unstable`. This also picked up some commits to the `defaultwrite` file, which will cause some synchronous behavior.

nfsv3commit.d

NFS commit operations can have a dramatic effect on write performance. The `nfsv3commit.d` script provides details of commits including size and time between commits.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

/* From /usr/include/nfs/nfs.h */
inline int UNSTABLE = 0;
int last[string];

dtrace:::BEGIN
{
        printf("Tracing NFSv3 writes and commits... Hit Ctrl-C to end.\n");
}

nfsv3:::op-write-start
/args[2]->stable == UNSTABLE/
{
        @write[args[1]->noi_curpath] = sum(args[2]->count);
}

nfsv3:::op-write-start
/args[2]->stable != UNSTABLE/
{
        @syncwrite[args[1]->noi_curpath] = sum(args[2]->count);
}

nfsv3:::op-commit-start
/(this->last = last[args[1]->noi_curpath])/
{
        this->delta = (timestamp - this->last) / 1000;
        @time[args[1]->noi_curpath] = quantize(this->delta);
}

nfsv3:::op-commit-start
{
```

continues

```
        @committed[args[1]->noi_curpath] = sum(args[2]->count);
        @commit[args[1]->noi_curpath] = quantize(args[2]->count / 1024);
        last[args[1]->noi_curpath] = timestamp;
}

dtrace:::END
{
        normalize(@write, 1024);
        normalize(@syncwrite, 1024);
        normalize(@committed, 1024);
        printf("\nCommited vs uncommited written Kbytes by path:\n\n");
        printf(" %-10s %-10s %-10s %s\n", "WRITE", "SYNCWRITE", "COMMITTED",
            "PATH");
        printa(" %@-10d %@-10d %@-10d %s\n", @write, @syncwrite, @committed);
        printf("\n\nCommit Kbytes by path:\n");
        printa(@commit);
        printf("\nTime between commits (us) by path:\n");
        printa(@time);
}
```

Script nfsv3commit.d

You can customize this script to show information by client instead of by path name.

This script has a small problem: The associative array called `last` is never freed. This means that the script can be run only for short durations (depends on how quickly different files are accessed), before the array will become so large that DTrace will drop data. If you see warning messages while running this script, it's been running too long.

Example

Two 1GB files were written to an NFSv3 share from two Solaris clients, using the default mount options and `open()` flags. The only difference between the clients is their kernel tunables: one uses the defaults, and the other is tuned.[6]

```
server# nfsv3commit.d
Tracing NFSv3 writes and commits... Hit Ctrl-C to end.
^C

Commited vs uncommited written Kbytes by path:

 WRITE      SYNCWRITE  COMMITTED  PATH
 1048576    0          1048576    /export/fs1/tuned-client
 1048576    0          1048576    /export/fs1/untuned-client


Commit Kbytes by path:
```

6. `tune_t_fsflushr=5` and `autoup=300` were set in `/etc/system`, causing the client to scan and flush dirty memory pages less frequently.

```
  /export/fs1/tuned-client
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@                     1
            1024 |                                         0
            2048 |                                         0
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@                     1
         1048576 |                                         0

  /export/fs1/untuned-client
           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@                                        2
            2048 |@@@@@@@@@@@@@@                           20
            4096 |@@@@@@@@                                 11
            8192 |@@@@@@@@@@@                              16
           16384 |@                                        2
           32768 |@@                                       3
           65536 |@                                        1
          131072 |@                                        1
          262144 |@                                        1
          524288 |                                         0

Time between commits (us) by path:

  /export/fs1/tuned-client
           value  ------------- Distribution ------------- count
          524288 |                                         0
         1048576 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
         2097152 |                                         0

  /export/fs1/untuned-client
           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@                         22
          262144 |@@@@@@@@                                 11
          524288 |@@@@@@@@@@@                              15
         1048576 |@                                        2
         2097152 |@@@                                      4
         4194304 |@                                        1
         8388608 |                                         0
        16777216 |@                                        1
        33554432 |                                         0
```

The runtime for creating the 1GB files on the clients was noticeably different, with the tuned client completing about four times quicker. The reason becomes clear with DTrace: The tuned client is making fewer, larger, less-frequent commits, whereas the untuned client is committing smaller sizes and more frequently.

A large time between commits isn't always because of tuning: The client may simply have stopped writing to the file for a while.

nfsv3errors.d

One of the network issue types we mentioned in Table 7-1 at the start of this chapter was errors. It may sound obvious to check for errors, but this is sometimes overlooked, especially if the system tools don't show them clearly to start with. While writing a DTrace script to examine NFS errors, we created known NFS errors to test the script. We also ran the supplied system tool `nfsstat`:

```
server# nfsstat
Server rpc:
Connection oriented:
calls      badcalls   nullrecv   badlen     xdrcall    dupchecks  dupreqs
899043     0          0          0          0          765544     0
Connectionless:
calls      badcalls   nullrecv   badlen     xdrcall    dupchecks  dupreqs
0          0          0          0          0          0          0

Server NFSv2:
calls     badcalls
0         0

Server NFSv3:
calls     badcalls
898959    0
[...]
Version 3: (897963 calls)
null        getattr     setattr     lookup      access      readlink
42 0%       4660 0%     4344 0%     1848 0%     1542 0%     0 0%
read        write       create      mkdir       symlink     mknod
121088 13%  759873 84%  857 0%      46 0%       5 0%        0 0%
remove      rmdir       rename      link        readdir     readdirplus
336 0%      23 0%       0 0%        0 0%        0 0%        60 0%
fsstat      fsinfo      pathconf    commit
47 0%       42 0%       704 0%      2446 0%
[...]
```

Even knowing what our errors were, we couldn't find them identified by the output of `nfsstat`. And even if we could, they would be single statistics without further information, such as which client encountered the error, what file it was for, and so on.

The `nfsv3errors.d` script traces NFSv3 errors as they occur, with client and filename information.

Script

The nfs provider makes this a very simple script. The bulk of code declares an associative array to translate from error codes to strings, based on the defines in `/usr/include/sys/nfs.h`:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

```

```
dtrace:::BEGIN
{
        /* See NFS3ERR_* in /usr/include/nfs/nfs.h */
        nfs3err[0] = "NFS3_OK";
        nfs3err[1] = "PERM";
        nfs3err[2] = "NOENT";
        nfs3err[5] = "IO";
        nfs3err[6] = "NXIO";
        nfs3err[13] = "ACCES";
        nfs3err[17] = "EXIST";
        nfs3err[18] = "XDEV";
        nfs3err[19] = "NODEV";
        nfs3err[20] = "NOTDIR";
        nfs3err[21] = "ISDIR";
        nfs3err[22] = "INVAL";
        nfs3err[27] = "FBIG";
        nfs3err[28] = "NOSPC";
        nfs3err[30] = "ROFS";
        nfs3err[31] = "MLINK";
        nfs3err[63] = "NAMETOOLONG";
        nfs3err[66] = "NOTEMPTY";
        nfs3err[69] = "DQUOT";
        nfs3err[70] = "STALE";
        nfs3err[71] = "REMOTE";
        nfs3err[10001] = "BADHANDLE";
        nfs3err[10002] = "NOT_SYNC";
        nfs3err[10003] = "BAD_COOKIE";
        nfs3err[10004] = "NOTSUPP";
        nfs3err[10005] = "TOOSMALL";
        nfs3err[10006] = "SERVERFAULT";
        nfs3err[10007] = "BADTYPE";
        nfs3err[10008] = "JUKEBOX";

        printf(" %-18s %5s %-12s %-16s %s\n", "NFSv3 EVENT", "ERR", "CODE",
            "CLIENT", "PATHNAME");
}

nfsv3:::op-*-done
/args[2]->status != 0/
{
        this->err = args[2]->status;
        this->str = nfs3err[this->err] != NULL ? nfs3err[this->err] : "?";
        printf(" %-18s %5d %-12s %-16s %s\n", probename, this->err,
            this->str, args[0]->ci_remote, args[1]->noi_curpath);
}
```

Script nfsv3errors.d

Example

The first error caught was ACCES (that spelling is from the `nfs.h` file), because the 192.168.1.109 client attempted to enter a directory it did not have permissions for. The remaining errors occurred because that client was reading a file that was deleted, causing outstanding reads to error as the file handle had became stale.

```
server# nfsv3errors.d
NFSv3 EVENT          ERR CODE          CLIENT            PATHNAME
op-lookup-done        13 ACCES         192.168.1.109     /export/fs1/secret
op-read-done          70 STALE         192.168.1.109     <unknown>
op-read-done          70 STALE         192.168.1.109     <unknown>
```

continues

```
op-read-done            70 STALE         192.168.1.109    <unknown>
op-read-done            70 STALE         192.168.1.109    <unknown>
op-read-done            70 STALE         192.168.1.109    <unknown>
[...]
```

nfsv3fbtrws.d

Should the nfs provider not be available or if you want customization beyond what the NFS provider can do, the fbt provider can be used. As a demonstration of this, the `nfsv3rwsnoop.d` script was rewritten to use the fbt provider. Since it now traces kernel functions directly, it is not expected to execute without adjustments to match the operating system kernel you are using.

This script was also rewritten to avoid later DTrace features, such as the `inet*()` functions to convert IP addresses to strings, to demonstrate ways these can be accomplished if those later features are not available.

Script

Compare the length and complexity of this script with the nfsv3 provider-based `nfsv3rwsnoop.d` script. With fbt, DTrace makes it possible; with stable providers, DTrace makes it both possible and easy.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-10s %6s %s\n", "TIME(us)",
            "CLIENT", "OP", "OFFSET(KB)", "BYTES", "PATHNAME");
}

fbt::rfs3_read:entry
{
        self->in_rfs3 = 1;
        /* args[0] is READ3args */
        self->offset = args[0]->offset / 1024;
        self->count = args[0]->count;
        self->req = args[3];
        self->dir = "R";
}

fbt::rfs3_write:entry
{
        self->in_rfs3 = 1;
        /* args[0] is WRITE3args */
        self->offset = args[0]->offset / 1024;
        self->count = args[0]->count;
        self->req = args[3];
        self->dir = "W";
}

/* trace nfs3_fhtovp() to retrieve the vnode_t */
fbt::nfs3_fhtovp:return
```

```
/self->in_rfs3/
{
        this->vp = args[1];
    this->socket = (struct sockaddr_in
             *)self->req->rq_xprt->xp_xpc.xpc_rtaddr.buf;
        /* DTrace 1.0: no inet functions, no this->strings */
        this->a = (uint8_t *)&this->socket->sin_addr.S_un.S_addr;
        self->addr1 = strjoin(lltostr(this->a[0] + 0ULL), strjoin(".",
            strjoin(lltostr(this->a[1] + 0ULL), ".")));
        self->addr2 = strjoin(lltostr(this->a[2] + 0ULL), strjoin(".",
            lltostr(this->a[3] + 0ULL)));
        self->address = strjoin(self->addr1, self->addr2);

        printf("%-16d %-18s %2s %-10d %6d %s\n", timestamp / 1000,
            self->address, self->dir, self->offset, self->count,
            this->vp->v_path != NULL ? stringof(this->vp->v_path) : "<?>");

        self->addr1 = 0;
        self->addr2 = 0;
        self->address = 0;
        self->dir = 0;
        self->req = 0;
        self->offset = 0;
        self->count = 0;
        self->in_rfs3 = 0;
}
```

Script nfsv3fbtrws.d

Because we're using the fbt provider, the script is highly dependent on the source implementation of NFS; in this case, it was written for a particular version of the OpenSolaris kernel.

Examples

Two examples follow.

Tracing NFSv3 I/O with fbt

```
server# nfsv3fbtrws.d
TIME(us)         CLIENT             OP OFFSET(KB)  BYTES PATHNAME
762366360517     192.168.110.3       R 64          32768 /export/fs11/50g-a-128k
762366348344     192.168.110.3       R 64          32768 /export/fs11/500m-cl-128k
762366360452     192.168.110.3       R 32          32768 /export/fs11/50g-a-128k
762366360522     192.168.110.3       R 160         32768 /export/fs11/50g-a-128k
762366348287     192.168.110.3       R 96          32768 /export/fs11/500m-cl-128k
762366359851     192.168.110.3       R 4           28672 /export/fs11/50g-a-128k
762366348340     192.168.110.3       R 160         32768 /export/fs11/500m-cl-128k
762366348260     192.168.110.3       R 32          32768 /export/fs11/500m-cl-128k
762366349761     192.168.110.3       R 0            4096 /export/fs11/50g-a-128k
762378489720     192.168.110.3       W 1536        32768 /export/fs11/test
762378490160     192.168.110.3       W 1600        32768 /export/fs11/test
762378490976     192.168.110.3       W 1696        32768 /export/fs11/test
762378491763     192.168.110.3       W 1792        32768 /export/fs11/test
[...]
```

The output is the same as the `nfsv3rwsnoop.d` script, as intended.

fbt Is "Unstable." Now this script was executed on a recent version of Solaris instead of OpenSolaris:

```
server# nfsv3fbtrws.d
dtrace: failed to compile script nfsv3fbtrws.d: line 12: probe description fbt::rfs3_
read:entry does not match any probes
```

This is an example of fbt's "unstable" interface. On this Solaris version, the `rfs3_read()` function has a different name. For this script to execute, it would need to be adjusted to match the kernel functions used by this Solaris version.

NFSv4 Scripts

The NFSv4 protocol includes features such as compound operations, allowing clients to group together NFS operations for improved performance. The scripts in this section are for tracing NFSv4 events on the NFS server. For NFSv4 client-side tracing, see Chapter 5.

A stable provider exists for NFSv4 with an almost identical interface to the NFSv3 provider. Because of this, many of the previous NFSv3 scripts require only small changes to work on NFSv4. Rather than repeating largely identical scripts, examples, and descriptions, here we will show only the changes. To see full descriptions and examples, refer to the earlier "NFSv3 Scripts" section.

Most of these scripts use the nfsv4 provider, which is fully documented in the nfsv4 provider section of the DTrace Guide.[7] It is currently available in OpenSolaris[8] and Solaris Nevada.[9] Listing the nfsv4 probes on Solaris Nevada, circa June 2010, yields the following:

```
solaris# dtrace -ln nfsv4:::
   ID   PROVIDER    MODULE                          FUNCTION NAME
11212      nfsv4    nfssrv                     rfs4_dispatch null-done
11213      nfsv4    nfssrv                     rfs4_dispatch null-start
11220      nfsv4    nfssrv                   rfs4_op_readdir op-readdir-done
11221      nfsv4    nfssrv                   rfs4_op_readdir op-readdir-start
11224      nfsv4    nfssrv                 rfs4_do_cb_recall cb-recall-done
11225      nfsv4    nfssrv                 rfs4_do_cb_recall cb-recall-start
11230      nfsv4    nfssrv                     rfs4_op_lockt op-lockt-done
11261      nfsv4    nfssrv                     rfs4_op_lockt op-lockt-start
11262      nfsv4    nfssrv                     rfs4_op_locku op-locku-done
11263      nfsv4    nfssrv                     rfs4_op_locku op-locku-start
11264      nfsv4    nfssrv                      rfs4_op_lock op-lock-done
```

7. *http://wikis.sun.com/display/DTrace/nfsv4+Provider*
8. PSARC 2007/665, CR 6635086, was integrated into Solaris Nevada in December 2007 (snv_80).
9. It is also shipped as part of the Oracle Sun ZFS Storage Appliance, where it powers NFSv4 Analytics.

```
  11265      nfsv4    nfsssrv                   rfs4_op_lock op-lock-start
  11266      nfsv4    nfsssrv                  rfs4_op_close op-close-done
  11267      nfsv4    nfsssrv                  rfs4_op_close op-close-start
  11268      nfsv4    nfsssrv    rfs4_op_setclientid_confirm op-setclientid-confirm-done
  11269      nfsv4    nfsssrv    rfs4_op_setclientid_confirm op-setclientid-confirm-start
  11270      nfsv4    nfsssrv            rfs4_op_setclientid op-setclientid-done
  11271      nfsv4    nfsssrv            rfs4_op_setclientid op-setclientid-start
  11272      nfsv4    nfsssrv         rfs4_op_open_downgrade op-open-downgrade-done
  11273      nfsv4    nfsssrv         rfs4_op_open_downgrade op-open-downgrade-start
  11274      nfsv4    nfsssrv           rfs4_op_open_confirm op-open-confirm-done
  11275      nfsv4    nfsssrv           rfs4_op_open_confirm op-open-confirm-start
  11276      nfsv4    nfsssrv                   rfs4_op_open op-open-done
  11277      nfsv4    nfsssrv                   rfs4_op_open op-open-start
  11282      nfsv4    nfsssrv                  rfs4_compound compound-done
  11283      nfsv4    nfsssrv                  rfs4_compound compound-start
  11284      nfsv4    nfsssrv                  rfs4_op_write op-write-done
  11285      nfsv4    nfsssrv                  rfs4_op_write op-write-start
  11286      nfsv4    nfsssrv                rfs4_op_nverify op-nverify-done
  11287      nfsv4    nfsssrv                rfs4_op_nverify op-nverify-start
  11288      nfsv4    nfsssrv                 rfs4_op_verify op-verify-done
  11289      nfsv4    nfsssrv                 rfs4_op_verify op-verify-start
  11290      nfsv4    nfsssrv                rfs4_op_setattr op-setattr-done
  11292      nfsv4    nfsssrv                rfs4_op_setattr op-setattr-start
  11293      nfsv4    nfsssrv                 rfs4_op_savefh op-savefh-done
  11294      nfsv4    nfsssrv                 rfs4_op_savefh op-savefh-start
  11295      nfsv4    nfsssrv              rfs4_op_restorefh op-restorefh-done
  11296      nfsv4    nfsssrv              rfs4_op_restorefh op-restorefh-start
  11299      nfsv4    nfsssrv                 rfs4_op_rename op-rename-done
  11301      nfsv4    nfsssrv                 rfs4_op_rename op-rename-start
  11302      nfsv4    nfsssrv                 rfs4_op_remove op-remove-done
  11305      nfsv4    nfsssrv                 rfs4_op_remove op-remove-start
  11306      nfsv4    nfsssrv      rfs4_op_release_lockowner op-release-lockowner-done
  11307      nfsv4    nfsssrv      rfs4_op_release_lockowner op-release-lockowner-start
  11308      nfsv4    nfsssrv               rfs4_op_readlink op-readlink-done
  11310      nfsv4    nfsssrv               rfs4_op_readlink op-readlink-start
  11311      nfsv4    nfsssrv              rfs4_op_putrootfh op-putrootfh-done
  11312      nfsv4    nfsssrv              rfs4_op_putrootfh op-putrootfh-start
  11313      nfsv4    nfsssrv                  rfs4_op_putfh op-putfh-done
  11314      nfsv4    nfsssrv                  rfs4_op_putfh op-putfh-start
  11315      nfsv4    nfsssrv               rfs4_op_putpubfh op-putpubfh-done
  11317      nfsv4    nfsssrv               rfs4_op_putpubfh op-putpubfh-start
  11318      nfsv4    nfsssrv                   rfs4_op_read op-read-done
  11319      nfsv4    nfsssrv                   rfs4_op_read op-read-start
  11320      nfsv4    nfsssrv               rfs4_op_openattr op-openattr-done
  11321      nfsv4    nfsssrv               rfs4_op_openattr op-openattr-start
  11322      nfsv4    nfsssrv                rfs4_op_lookupp op-lookupp-done
  11323      nfsv4    nfsssrv                rfs4_op_lookupp op-lookupp-start
  11324      nfsv4    nfsssrv                 rfs4_op_lookup op-lookup-done
  11325      nfsv4    nfsssrv                 rfs4_op_lookup op-lookup-start
  11327      nfsv4    nfsssrv                   rfs4_op_link op-link-done
  11328      nfsv4    nfsssrv                   rfs4_op_link op-link-start
  11329      nfsv4    nfsssrv                  rfs4_op_getfh op-getfh-done
  11333      nfsv4    nfsssrv                  rfs4_op_getfh op-getfh-start
  11334      nfsv4    nfsssrv                rfs4_op_getattr op-getattr-done
  11335      nfsv4    nfsssrv                rfs4_op_getattr op-getattr-start
  11337      nfsv4    nfsssrv            rfs4_op_delegreturn op-delegreturn-done
  11338      nfsv4    nfsssrv            rfs4_op_delegreturn op-delegreturn-start
  11339      nfsv4    nfsssrv             rfs4_op_delegpurge op-delegpurge-done
  11340      nfsv4    nfsssrv             rfs4_op_delegpurge op-delegpurge-start
  11341      nfsv4    nfsssrv                 rfs4_op_create op-create-done
  11342      nfsv4    nfsssrv                 rfs4_op_create op-create-start
  11343      nfsv4    nfsssrv                 rfs4_op_commit op-commit-done
  11344      nfsv4    nfsssrv                 rfs4_op_commit op-commit-start
  11345      nfsv4    nfsssrv                 rfs4_op_access op-access-done
  11348      nfsv4    nfsssrv                 rfs4_op_access op-access-start
  11349      nfsv4    nfsssrv                rfs4_op_secinfo op-secinfo-done
  11350      nfsv4    nfsssrv                rfs4_op_secinfo op-secinfo-start
```

Both the compound operation can be traced with the `compound-start` and `compound-done` probes, as well as the individual NFS operations with the `op-*-start` and `op-*-done` probes. Each provide arguments for the operation, including client address and filename (when appropriate). The previous listing also highlights the locations of the probes in the nfssrv kernel module by showing the kernel functions that contain them (`FUNCTION` column). These can be treated as starting points if you need to examine the source code.

If the nfsv4 provider is not available, the fbt provider can be used, bearing in mind that fbt-based scripts may only execute on the kernel version they were written for. Finding the right kernel functions to trace using the fbt provider can sometimes be a challenge. However, if the kernel version you are using is anything like the Solaris Nevada version shown previously, the function names may be similar to the operation names, making them easy to find. An example of fbt provider tracing of NFSv4 is included in this section: `nfsv4deleg.d`.

nfsv4rwsnoop.d

This script traces NFSv4 reads and writes live, with client, I/O size, and path name details.

The script is identical to the `nfsv3rwsnoop.d` script, with the following different lines:

```
12  nfsv4:::op-read-start
19  nfsv4:::op-write-start
23              args[2]->data_len, args[1]->noi_curpath);
```

nfsv4ops.d

This script shows NFSv4 operation counts and prints a summary every five seconds. The script is identical to the `nfsv3ops.d` script, with the following different lines:

```
7           trace("Tracing NFSv4 operations... Interval 5 secs.\n");
10  nfsv4:::op-*-start
```

nfsv4fileio.d

This script summarizes NFSv4 read and write bytes by filename. The script is identical to the `nfsv3fileio.d` script, with the following different lines:

```
10  nfsv4:::op-read-done
12          @readbytes[args[1]->noi_curpath] = sum(args[2]->data_len);
15  nfsv4:::op-write-done
17          @writebytes[args[1]->noi_curpath] = sum(args[2]->count);
```

nfsv4rwtime.d

This script shows NFSv4 read and write latency and top clients and files. The script is identical to the `nfsv3rwtime.d` script, with the following different lines:

```
12  nfsv4:::op-read-start,
13  nfsv4:::op-write-start
18  nfsv4:::op-read-done,
19  nfsv4:::op-write-done
32          printf("NFSv4 read/write distributions (us):\n");
35          printf("\nNFSv4 read/write by host (total us):\n");
39          printf("\nNFSv4 read/write top %d files (total us):\n", TOP_FILES);
```

nfsv4syncwrite.d

This script identifies synchronous write workloads. The script is identical to the `nfsv3syncwrite.d` script, with the following different lines:

```
7           /* See /usr/include/nfs/nfs4_kprot.h */
11          printf("Tracing NFSv4 writes and commits... Hit Ctrl-C to end.\n");
14  nfsv4:::op-write-start
19  nfsv4:::op-commit-start
```

nfsv4commit.d

This script summarizes details of NFSv4 commit operations. The script is identical to the `nfsv3commit.d` script, with the following different lines:

```
 5  /* From /usr/include/nfs/nfs4_kprot.h */
11          printf("Tracing NFSv4 writes and commits... Hit Ctrl-C to end.\n");
14  nfsv4:::op-write-start
17          @write[args[1]->noi_curpath] = sum(args[2]->data_len);
20  nfsv4:::op-write-start
23          @syncwrite[args[1]->noi_curpath] = sum(args[2]->data_len);
26  nfsv4:::op-commit-start
33  nfsv4:::op-commit-start
```

nfsv4errors.d

This script traces NFSv4 errors live, with details. This script is very different from the NFSv3 version, and so is shown in its entirety here.

Script

The `nfsv4errors.d` script is similar in operation to `nfsv3errors.d` but with a different error translation table and logic to skip NFSv4 lookup `SAME` errors.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        /* See NFS4ERR_* in /usr/include/nfs/nfs4_kprot.h */
        nfs4err[0] = "NFS4_OK";
        nfs4err[1] = "PERM";
        nfs4err[2] = "NOENT";
        nfs4err[5] = "IO";
        nfs4err[6] = "NXIO";
        nfs4err[13] = "ACCESS";
        nfs4err[17] = "EXIST";
        nfs4err[18] = "XDEV";
        nfs4err[20] = "NOTDIR";
        nfs4err[21] = "ISDIR";
        nfs4err[22] = "INVAL";
        nfs4err[27] = "FBIG";
        nfs4err[28] = "NOSPC";
        nfs4err[30] = "ROFS";
        nfs4err[31] = "MLINK";
        nfs4err[63] = "NAMETOOLONG";
        nfs4err[66] = "NOTEMPTY";
        nfs4err[69] = "DQUOT";
        nfs4err[70] = "STALE";
        nfs4err[10001] = "BADHANDLE";
        nfs4err[10003] = "BAD_COOKIE";
        nfs4err[10004] = "NOTSUPP";
        nfs4err[10005] = "TOOSMALL";
        nfs4err[10006] = "SERVERFAULT";
        nfs4err[10007] = "BADTYPE";
        nfs4err[10008] = "DELAY";
        nfs4err[10009] = "SAME";
        nfs4err[10010] = "DENIED";
        nfs4err[10011] = "EXPIRED";
        nfs4err[10012] = "LOCKED";
        nfs4err[10013] = "GRACE";
        nfs4err[10014] = "FHEXPIRED";
        nfs4err[10015] = "SHARE_DENIED";
        nfs4err[10016] = "WRONGSEC";
        nfs4err[10017] = "CLID_INUSE";
        nfs4err[10018] = "RESOURCE";
        nfs4err[10019] = "MOVED";
        nfs4err[10020] = "NOFILEHANDLE";
        nfs4err[10021] = "MINOR_VERS_MISMATCH";
        nfs4err[10022] = "STALE_CLIENTID";
        nfs4err[10023] = "STALE_STATEID";
        nfs4err[10024] = "OLD_STATEID";
        nfs4err[10025] = "BAD_STATEID";
        nfs4err[10026] = "BAD_SEQID";
        nfs4err[10027] = "NOT_SAME";
        nfs4err[10028] = "LOCK_RANGE";
        nfs4err[10029] = "SYMLINK";
        nfs4err[10030] = "RESTOREFH";
        nfs4err[10031] = "LEASE_MOVED";
        nfs4err[10032] = "ATTRNOTSUPP";
        nfs4err[10033] = "NO_GRACE";
        nfs4err[10034] = "RECLAIM_BAD";
        nfs4err[10035] = "RECLAIM_CONFLICT";
        nfs4err[10036] = "BADXDR";
        nfs4err[10037] = "LOCKS_HELD";
        nfs4err[10038] = "OPENMODE";
        nfs4err[10039] = "BADOWNER";
```

```
        nfs4err[10040] = "BADCHAR";
        nfs4err[10041] = "BADNAME";
        nfs4err[10042] = "BAD_RANGE";
        nfs4err[10043] = "LOCK_NOTSUPP";
        nfs4err[10044] = "OP_ILLEGAL";
        nfs4err[10045] = "DEADLOCK";
        nfs4err[10046] = "FILE_OPEN";
        nfs4err[10047] = "ADMIN_REVOKED";
        nfs4err[10048] = "CB_PATH_DOWN";

        printf(" %-18s %5s %-12s %-16s %s\n", "NFSv4 EVENT", "ERR", "CODE",
            "CLIENT", "PATHNAME");
}

nfsv4:::op-*-done
/args[2]->status != 0 && args[2]->status != 10009/
{
        this->err = args[2]->status;
        this->str = nfs4err[this->err] != NULL ? nfs4err[this->err] : "?";
        printf(" %-18s %5d %-12s %-16s %s\n", probename, this->err,
            this->str, args[0]->ci_remote, args[1]->noi_curpath);
}
```

Example

Here's an example of this script running on an NFS home directory server:

```
server# nfsv4errors.d
 NFSv4 EVENT      ERR CODE          CLIENT           PATHNAME
 op-lookup-done   2 NOENT         192.168.1.110  /export/home/bmc/.mozilla/firefox/j89zrwbl.default
 op-lookup-done   2 NOENT         192.168.1.110  /export/home/bmc
 op-lookup-done   2 NOENT         192.168.1.110  /export/home/bmc/.mozilla/firefox/j89zrwbl.default
 op-verify-done   10027 NOT_SAME 192.168.1.110  /export/home/bmc/.mozilla/firefox/j89zrwbl.default/
                                                 places.s...
 op-lookup-done   2 NOENT         192.168.1.110  /export/home/bmc/.mozilla/firefox/j89zrwbl.default
 op-lookup-done   2 NOENT         192.168.1.110  /export/home/bmc/.mozilla/firefox/j89zrwbl.default
 op-lookup-done   2 NOENT         192.168.1.109  /export/home/brendan/.mozilla/firefox/
                                                 tafu5y0e.default
 op-write-done    13 ACCESS       192.168.1.109  /export/home/brendan/.mozilla/firefox/
                                                 tafu5y0e.default/Cach...
```

nfsv4deleg.d

If the nfsv4 provider is unavailable, the fbt provider can be used. It can also extend the observability of the stable nfsv4 provider, albeit in an unstable manner. Here the fbt provider is used to examine NFSv4 delegation events, beyond what is available in the nfsv4 provider. Because the fbt provider is tracing kernel functions directly, this script is not expected to execute without adjustments to match the operating system kernel you are using.

Script

Writing this script involved applying a known (assumed) workload to trigger NFSv4 write delegations and using DTrace to frequency count all NFSv4 functions that fired. This identified many functions containing the word *delegation* or *deleg*; these functions were then read from the NFSv4 source to further understand them and to see what arguments they provided. This script traces the functions that were found and the arguments that were identified in the source code:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        deleg[0] = "none";
        deleg[1] = "read";
        deleg[2] = "write";
        deleg[-1] = "any";

        printf("Tracing NFSv4 delegation events...\n");
        printf("%-21s %-20s %s\n", "TIME", "EVENT", "DETAILS");
}

fbt::rfs4_grant_delegation:entry
{
        this->path = stringof(args[1]->rs_finfo->rf_vp->v_path);
        this->client = args[1]->rs_owner->ro_client->rc_clientid;
        this->type = deleg[arg0] != NULL ? deleg[arg0] : "<?>";
        printf("%-21Y %-20s %-8s %s\n", walltimestamp, "Grant Delegation",
            this->type, this->path);
}

fbt::rfs4_recall_deleg:entry
{
        this->path = stringof(args[0]->rf_vp->v_path);
        printf("%-21Y %-20s %-8s %s\n", walltimestamp, "Recall Delegation",
            ".", this->path);
}

fbt::rfs4_deleg_state_expiry:entry
{
        this->dsp = (rfs4_deleg_state_t *)arg0;
        this->path = stringof(this->dsp->rds_finfo->rf_vp->v_path);
        printf("%-21Y %-20s %-8s %s\n", walltimestamp, "Delegation Expiry",
            ".", this->path);
}
```

Script nfsv4deleg.d

Example

Here two clients wrote in turn to the same file, `newfile`:

```
server# nfsv4deleg.d
Tracing NFSv4 delegation events...
TIME                  EVENT                DETAILS
2010 Jan 12 05:17:59  Grant Delegation     any      /export/fs1/newfile
2010 Jan 12 05:18:05  Recall Delegation    .        /export/fs1/newfile
2010 Jan 12 05:18:06  Grant Delegation     none     /export/fs1/newfile
2010 Jan 12 05:18:50  Delegation Expiry    .        /export/fs1/newfile
```

The next step would be to enhance the script to include client information, such as their IP address.

CIFS Scripts

The Common Internet File System (CIFS) protocol, also known as Server Message Block (SMB), is commonly used by Microsoft Windows clients. It can be examined using DTrace in a similar way and for similar reasons to the NFS protocol, as shown in the previous section.

The scripts in this section are for tracing CIFS events on the CIFS server. For CIFS client-side tracing, see Chapter 5.

Most of these scripts use the smb provider, which was developed for and included in the Oracle Sun ZFS Storage Appliance. It's not currently available elsewhere; however, we hope that it has been included in Solaris (at least) by the time you are reading this.[10] Listing the smb provider probes yields the following:

```
# dtrace -ln smb:::
   ID   PROVIDER        MODULE                         FUNCTION NAME
  100        smb        smbsrv              smb_post_write_raw op-WriteRaw-done
  101        smb        smbsrv               smb_pre_write_raw op-WriteRaw-start
  102        smb        smbsrv             smb_post_write_andx op-WriteX-done
  103        smb        smbsrv              smb_pre_write_andx op-WriteX-start
  104        smb        smbsrv       smb_post_write_and_unlock op-WriteAndUnlock-done
  105        smb        smbsrv        smb_pre_write_and_unlock op-WriteAndUnlock-start
  106        smb        smbsrv        smb_post_write_and_close op-WriteAndClose-done
  107        smb        smbsrv         smb_pre_write_and_close op-WriteAndClose-start
  108        smb        smbsrv                  smb_post_write op-Write-done
  109        smb        smbsrv                   smb_pre_write op-Write-start
  114        smb        smbsrv     smb_post_unlock_byte_range op-UnlockByteRange-done
  115        smb        smbsrv      smb_pre_unlock_byte_range op-UnlockByteRange-start
  116        smb        smbsrv       smb_post_tree_disconnect op-TreeDisconnect-done
  117        smb        smbsrv        smb_pre_tree_disconnect op-TreeDisconnect-start
[...]
  149        smb        smbsrv              smb_post_read_andx op-ReadX-done
  150        smb        smbsrv               smb_pre_read_andx op-ReadX-start
  151        smb        smbsrv               smb_post_read_raw op-ReadRaw-done
  152        smb        smbsrv                smb_pre_read_raw op-ReadRaw-start
  153        smb        smbsrv          smb_post_lock_and_read op-LockAndRead-done
  154        smb        smbsrv           smb_pre_lock_and_read op-LockAndRead-start
  155        smb        smbsrv                   smb_post_read op-Read-done
  156        smb        smbsrv                    smb_pre_read op-Read-start
[...]
  171        smb        smbsrv              smb_post_open_andx op-OpenX-done
  172        smb        smbsrv               smb_pre_open_andx op-OpenX-start
  173        smb        smbsrv                   smb_post_open op-Open-done
  174        smb        smbsrv                    smb_pre_open op-Open-start
[...]
```

10. This is likely; other providers including ip and tcp also began life in the Oracle Sun ZFS Storage Appliance before being ported elsewhere: first to Solaris Nevada and, from that, OpenSolaris.

```
   237        smb        smbsrv          smb_post_transaction op-Transaction-done
   238        smb        smbsrv             smb_pre_transaction op-Transaction-start
[...]
   241        smb        smbsrv                  smb_post_close op-Close-done
   242        smb        smbsrv                   smb_pre_close op-Close-start
```

The previous output has been truncated to include just the read, write, open, transaction, and close probes. The full listing of the smb provider probes would span a few pages.

CIFS operations can be traced with the `op-*-start` and `op-*-done` probes. Each provide arguments for the operation, including client address and filename (when appropriate). The previous listing also highlights the locations of the probes in the smbsrv kernel module, by showing the kernel functions that contain them (`FUNCTION` column). These can be treated as starting points should you need to examine the source code.

If the smb provider is not available, the pid provider or fbt provider can be used; pid can be used if the CIFS software is user-land based (for example, Samba[11]), and fbt can be used if it is kernel-based (for example, smbsrv). Bear in mind that pid or fbt-based scripts may only execute on the software version they were written for. An example of fbt provider tracing of CIFS (smbsrv) is included in this section: `cifsfbtnofile.d`.

cifsrwsnoop.d

This script traces CIFS reads and writes live, with client, I/O size, and path name details (if available).

Script

The script probes CIFS Read and ReadX (large read—supports 64-bit offsets), Write, and WriteX operations:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-10s %6s %s\n", "TIME(us)",
            "CLIENT", "OP", "OFFSET(KB)", "BYTES", "PATHNAME");
}

```

11. *www.samba.org*

```
smb:::op-Read-done, smb:::op-ReadX-done
{
        this->dir = "R";
}

smb:::op-Write-done, smb:::op-WriteX-done
{
        this->dir = "W";
}

smb:::op-Read-done, smb:::op-ReadX-done,
smb:::op-Write-done, smb:::op-WriteX-done
{
        printf("%-16d %-18s %2s %-10d %6d %s\n", timestamp / 1000,
            args[0]->ci_remote, this->dir, args[2]->soa_offset / 1024,
            args[2]->soa_count, args[1]->soi_curpath);
}
```

Script cifsrwsnoop.d

Unlike the `nfsv3rwsnoop.d` script, this is measuring when the events have completed—their "done" probes fire.

Example

Here a 1MB file was created from a remote client to a CIFS share:

```
server# cifsrwsnoop.d
TIME(us)          CLIENT             OP OFFSET(KB)  BYTES PATHNAME
999489329684      192.168.2.51        W 0           61440 /export/fs8/1m-file
999489330579      192.168.2.51        W 60          61440 /export/fs8/1m-file
999489331504      192.168.2.51        W 120         61440 /export/fs8/1m-file
999489332372      192.168.2.51        W 180         61440 /export/fs8/1m-file
999489334219      192.168.2.51        W 300         61440 /export/fs8/1m-file
999489333319      192.168.2.51        W 240         61440 /export/fs8/1m-file
999489335192      192.168.2.51        W 360         61440 /export/fs8/1m-file
999489336098      192.168.2.51        W 420         61440 /export/fs8/1m-file
999489337041      192.168.2.51        W 480         61440 /export/fs8/1m-file
999489337898      192.168.2.51        W 540         61440 /export/fs8/1m-file
999489338837      192.168.2.51        W 600         61440 /export/fs8/1m-file
999489339822      192.168.2.51        W 660         61440 /export/fs8/1m-file
999489340787      192.168.2.51        W 720         61440 /export/fs8/1m-file
999489341706      192.168.2.51        W 780         61440 /export/fs8/1m-file
999489342650      192.168.2.51        W 840         61440 /export/fs8/1m-file
999489343565      192.168.2.51        W 900         61440 /export/fs8/1m-file
999489344430      192.168.2.51        W 960         61440 /export/fs8/1m-file
999489344664      192.168.2.51        W 1020         4096 /export/fs8/1m-file
```

Note that the output is shuffled just a little; examine the `TIME(us)` column. This is expected behavior for DTrace scripts that print live output from multiple CPU buffers and is why the `TIME` column is printed: for postsorting.

cifsops.d

This script shows CIFS operation counts and prints a summary every five seconds.

Script

All smb provider events are traced and counted in the @ops aggregation, which is printed and then cleared every five seconds:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing CIFS operations... Interval 5 secs.\n");
}

smb:::op-*
{
        @ops[args[0]->ci_remote, probename] = count();
}

profile:::tick-5sec,
dtrace:::END
{
        printf("\n   %-32s %-30s %8s\n", "Client", "Operation", "Count");
        printa("   %-32s %-30s %@8d\n", @ops);
        trunc(@ops);
}
```

Script cifsops.d

Example

This script quickly identifies CIFS clients, with an idea of their usage:

```
server# cifsops.d
Tracing CIFS operations... Interval 5 secs.

   Client                           Operation                         Count
   192.168.2.51                     op-Close-done                         2
   192.168.2.51                     op-Close-start                        2
   192.168.2.51                     op-NtCreateX-done                     2
   192.168.2.51                     op-NtCreateX-start                    2
   192.168.2.51                     op-Transaction2-done                  4
   192.168.2.51                     op-Transaction2-start                 4
   192.168.2.51                     op-WriteX-done                       18
   192.168.2.51                     op-WriteX-start                      18

   Client                           Operation                         Count
   192.168.2.51                     op-NtCreateX-done                     1
   192.168.2.51                     op-NtCreateX-start                    1
   192.168.2.51                     op-Transaction2-done                  4
   192.168.2.51                     op-Transaction2-start                 4
   192.168.2.51                     op-ReadX-done                     18113
   192.168.2.51                     op-ReadX-start                    18113
```

```
    Client                          Operation                          Count
    192.168.2.51                    op-ReadX-done                      18549
    192.168.2.51                    op-ReadX-start                     18549
```

In the first five-second output, 192.168.2.51 wrote a file and performed some other operations; it then began a read workload of more than 18,000 events every five seconds (3,600 IOPS).

cifsfileio.d

This script summarizes CIFS read and write bytes by filename.

Script

We can track file path names, as well as the number of bytes read or written per file.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... Hit Ctrl-C to end.\n");
}

smb:::op-Read-done, smb:::op-ReadX-done
{
        @readbytes[args[1]->soi_curpath] = sum(args[2]->soa_count);
}

smb:::op-Write-done, smb:::op-WriteX-done
{
        @writebytes[args[1]->soi_curpath] = sum(args[2]->soa_count);
}

dtrace:::END
{
        printf("\n%12s %12s  %s\n", "Rbytes", "Wbytes", "Pathname");
        printa("%@12d %@12d  %s\n", @readbytes, @writebytes);
}
```

Script cifsfileio.d

Example

This example uses a known workload.

Here we wrote a 100MB file and read a 10MB file to confirm byte counts are measured correctly:

```
server# cifsfileio.d
Tracing... Hit Ctrl-C to end.
^C
```

continues

```
    Rbytes       Wbytes  Pathname
         0    104857600  /export/fs8/100m
  10485760            0  /export/fs8/10m
```

cifsrwtime.d

This script shows CIFS read and write latency and top clients and files.

Script

In the current implementation of CIFS for which the smb provider is written, the operation `start` probes fire in the same thread as the `done` probes. This allows timing between them to be tracked using thread-local variables (`self->`), instead of saving start times in an associative array keyed on a unique I/O ID.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

inline int TOP_FILES = 10;

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

smb:::op-Read-start, smb:::op-ReadX-start,
smb:::op-Write-start, smb:::op-WriteX-start
{
        /* currently the done event fires in the same thread as start */
        self->start = timestamp;
}

smb:::op-Read-done, smb:::op-ReadX-done   { this->dir = "read"; }
smb:::op-Write-done, smb:::op-WriteX-done { this->dir = "write"; }

smb:::op-Read-done, smb:::op-ReadX-done,
smb:::op-Write-done, smb:::op-WriteX-done
/self->start/
{
        this->elapsed = timestamp - self->start;
        @rw[this->dir] = quantize(this->elapsed / 1000);
        @host[args[0]->ci_remote] = sum(this->elapsed);
        @file[args[1]->soi_curpath] = sum(this->elapsed);
        self->start = 0;
}

dtrace:::END
{
        printf("CIFS read/write distributions (us):\n");
        printa(@rw);

        printf("\nCIFS read/write by host (total us):\n");
        normalize(@host, 1000);
        printa(@host);

        printf("\nCIFS read/write top %d files (total us):\n", TOP_FILES);
        normalize(@file, 1000);
```

```
        trunc(@file, TOP_FILES);
        printa(@file);
}
```

Script cifsrwtime.d

Example

Here the `cifsrwtime.d` script measured the read and write latency of some test workloads:

```
server# cifsrwtime.d
Tracing... Hit Ctrl-C to end.
^C
CIFS read/write distributions (us):

  write
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |                                         1
              16 |@@                                       8
              32 |                                         1
              64 |@@@@@@@@@@@@@@@@@@@@                     89
             128 |@@@@@@@@@@@@@@@@@                        78
             256 |@                                        3
             512 |                                         0

  read
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        21036
              16 |@@@@@@@                                  4367
              32 |                                         170
              64 |                                         25
             128 |                                         1
             256 |                                         1
             512 |                                         0


CIFS read/write by host (total us):

  192.168.2.51                                                 409718

CIFS read/write top 10 files (total us):

  /export/fs8/10m-file                                          24981
  /export/fs8/100m                                             384737
```

Most of the reads were between 8 us and 15 us, suggesting hitting from file system cache. The writes took longer, 64 us to 127 us, which may be because of flushing to disk. DTrace can be used to confirm both of these scenarios by tracing into the local file system that CIFS is exporting.

cifserrors.d

This script traces CIFS errors live, with details.

Script

When CIFS operations fail, a struct `smb_error` contains various details:

```
typedef struct {
        uint32_t severity;
        uint32_t status;
        uint16_t errcls;
        uint16_t errcode;
} smb_error_t;
```

This script fetches the status code and prints it if it was unsuccessful. Some error code translation is also provided; however, there are hundreds of codes to translate from, so to keep this script short, only the most likely CIFS error codes are processed here:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        /*
         * These are some of over 500 NT_STATES_* error codes defined in
         * uts/common/smbsrv/ntstatus.h.  For more detail see MSDN and
         * ntstatus.h in the MS DDK.
         */
        ntstatus[0] = "SUCCESS";
        ntstatus[1] = "UNSUCCESSFUL";
        ntstatus[2] = "NOT_IMPLEMENTED";
        ntstatus[5] = "ACCESS_VIOLATION";
        ntstatus[15] = "NO_SUCH_FILE";
        ntstatus[17] = "END_OF_FILE";
        ntstatus[23] = "NO_MEMORY";
        ntstatus[29] = "ILLEGAL_INSTRUCTION";
        ntstatus[34] = "ACCESS_DENIED";
        ntstatus[50] = "DISK_CORRUPT_ERROR";
        ntstatus[61] = "DATA_ERROR";
        ntstatus[62] = "CRC_ERROR";
        ntstatus[68] = "QUOTA_EXCEEDED";
        ntstatus[127] = "DISK_FULL";
        ntstatus[152] = "FILE_INVALID";
        ntstatus[186] = "FILE_IS_A_DIRECTORY";
        ntstatus[258] = "FILE_CORRUPT_ERROR";
        ntstatus[259] = "NOT_A_DIRECTORY";
        ntstatus[291] = "FILE_DELETED";
        /* ...etc... */

        printf(" %-24s %3s %-19s %-16s %s\n", "CIFS EVENT", "ERR", "CODE",
            "CLIENT", "PATHNAME");
}

smb:::op-*-start, smb:::op-*-done
/(this->sr = (struct smb_request *)arg0) && this->sr->smb_error.status != 0/
{
        this->err = this->sr->smb_error.status;
        this->str = ntstatus[this->err] != NULL ? ntstatus[this->err] : "?";
```

```
43          printf(" %-24s %3d %-19s %-16s %s\n", probename, this->err,
44              this->str, args[0]->ci_remote, args[1]->soi_curpath);
45  }
```

Script cifserrors.d

Since the error status is not currently a member of the stable smb provider, it had to be fetched from outside the stable provider interface. This was achieved by accessing `arg0` on line 39, which for these probes is a `struct smb_request` pointer and is the input to the provider translators (as defined in `/usr/lib/dtrace/smb.d`) that turn it into `(conninfo_t *)args[0]` and `(smbopinfo_t *)args[1]`. By accessing it pretranslation (and casting it, since `arg0` is technically a `uint64_t`), we can access any internal variable from the kernel, including the error status as is checked on line 39 and saved on line 41. This also makes the script unstable—should the smbsrv kernel code change, the smb provider could also change such that `arg0` points to a different struct entirely, and the casting on line 39 would be invalid, causing the script to print invalid data as the error status. Be careful to double-check this (and all) unstable scripts before use.

Example

A couple of CIFS errors were caught while running this script:

```
server# cifserrors.d
 CIFS EVENT                ERR CODE                 CLIENT           PATHNAME
 op-Transaction2-done       15 NO_SUCH_FILE         192.168.2.51     <unknown>
 op-NtCreateX-done         259 NOT_A_DIRECTORY      192.168.2.51     <unknown>
```

The path name for these error operations was unavailable, and so `<unknown>` was printed. The path name printed by the CIFS provider is the local file system path name, which is retrieved from the vnode object. If the requested file doesn't exist, neither does a file system vnode, so the path name is `<unknown>`.

With more DTrace scripting of the CIFS implementation via the unstable fbt provider, the full path name could be retrieved as requested via the SMB protocol.

cifsfbtnofile.d

If the smb provider is not be available on your OS, you can still investigate CIFS operations using the fbt provider. To demonstrate fbt provider tracing, this script pulls out the requested path names for lookups that failed, which can be the sign of a misconfigured application. In the `cifserrors.d` script shown previously, these were showing up as `<unknown>` because there was no file system vnode for a missing file.

Since fbt is an unstable provider, this may work only on a particular version of the kernel smb module. For it to execute on other versions, it will need to be adjusted to match the kernel functions it traces.

Script

To develop this script, a known workload of failed lookups was applied while DTrace traced which functions were called. These functions were then examined in the source code to determine how to extract the path name, error, and client details from these function calls.

IPv4 and IPv6 addresses were printed using the `inet*()` functions, which may not be available on your version of DTrace; if so, rewrite using manual IP address translation as demonstrated earlier (`soconnect.d` section).

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        /* a few likely codes are included from ntstatus.h */
        ntstatus[0] = "SUCCESS";
        ntstatus[1] = "UNSUCCESSFUL";
        ntstatus[15] = "NO_SUCH_FILE";
        ntstatus[186] = "FILE_IS_A_DIR";

        printf(" %-16s %3s %-13s %s\n", "CLIENT", "ERR", "ERROR", "PATHNAME");
}

fbt::smb*find_first*:entry  { self->in_find_first = 1; }
fbt::smb*find_first*:return { self->in_find_first = 0; }

/* assume smb_odir_open() checks relevant path during find_entries */
fbt::smb_odir_open:entry
/self->in_find_first/
{
        self->sr = args[0];
        self->path = args[1];
}

/* assume smbsr_set_error() will set relevant error during find_entries */
fbt::smbsr_set_error:entry
/self->in_find_first/
{
        self->err = args[1]->status;
}

/* if an error was previously seen during find_entries, print cached details */
fbt::smb*find_entries:return
/self->sr && self->err/
{
        this->str = ntstatus[self->err] != NULL ? ntstatus[self->err] : "?";
        this->remote = self->sr->session->ipaddr.a_family == AF_INET ?
            inet_ntoa(&self->sr->session->ipaddr.au_addr.au_ipv4) :
            inet_ntoa6(&self->sr->session->ipaddr.au_addr.au_ipv6);
```

```
        printf(" %-16s %3d %-13s %s%s\n", this->remote, self->err, this->str,
            self->sr->tid_tree->t_sharename, stringof(self->path));
}

fbt::smb*find_entries:return
/self->sr/
{
        self->sr = 0; self->path = 0; self->err = 0;
}
```

Script cifsfbtnofile.d

Example

To test this script, a known workload was applied to read invalid filenames with a random component. The script picked up the errors correctly:

```
server# cifsfbtnofile.d
 CLIENT           ERR ERROR         PATHNAME
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_15627
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_17623
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_17755
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_14952
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_28375
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_3690
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_28705
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_29068
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_10849
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_12502
 192.168.2.51      15 NO_SUCH_FILE  rpool_fs8\whereismyfile_8457
[...]
```

Unlike the smb provider–based scripts, here we can see that the CIFS share name accessed rpool_fs8.

HTTP Scripts

The Hypertext Transfer Protocol (HTTP) is currently the most common protocol for Web browsers. Browsers such as Firefox support plug-ins that allow a certain degree of analysis of HTTP from the client. DTrace can extend HTTP analysis by examining it in the same context as browser and system execution, as well as server-side analysis.

Use DTrace to answer the following:

Client HTTP requests, by Web server, by URL

Inbound HTTP requests, by client, by URL

First-byte latency

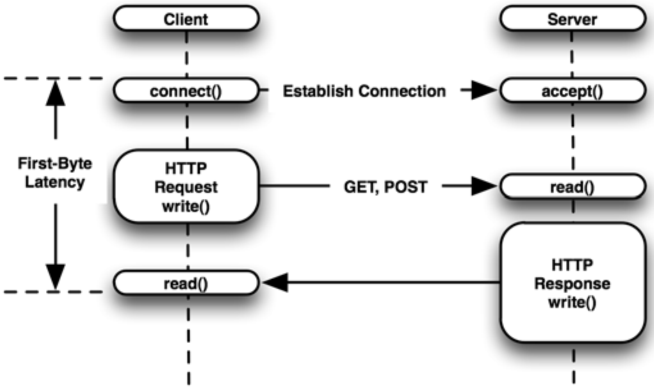


Figure 7-1 HTTP flow diagram

The HTTP protocol is usually served from user-land Web servers and processed by user-land Web browsers. Because of this, the scripts that follow will use user-level providers such as syscall and pid. The syscall provider allows the HTTP protocol to be examined at the socket level, and the pid provider allows complete instrumentation of processing in the user code, albeit an unstable interface that is dependent on the software implementation.

Since HTTP is a popular protocol to analyze, specific user-level stable providers have been written to provide HTTP-level events for easy tracing and analysis. So far, these have been implemented as stable USDT providers from a pluggable Apache module. David Pacheco of Oracle[12] and Ryan Matteson of Prefetch Technologies[13] have both independently written USDT-based mod_dtrace plug-ins for Apache. If you intend to DTrace HTTP on Apache, your options are as follows:

Choose a mod_dtrace plug-in to use

Write your own mod_dtrace plug-in

Use the pid provider on `httpd`

Try clever uses of the syscall provider

12. *http://blogs.sun.com/dap/entry/writing_a_dtrace_usdt_provider* discusses how to write USDT providers. This http provider is used by the Oracle Sun ZFS Storage Appliance.

13. *http://prefetch.net/projects/apache_modtrace* includes the `mod_dtrace.c` source and many ready-to-use scripts.

The interface for the existing mod_dtrace plug-in providers are slightly different. As an example, the following lists the probes for the Oracle http provider:

```
server# dtrace -ln 'http*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 9434   http7846     mod_dtrace.so            mod_dtrace_postrequest request-done
 9435   http7846     mod_dtrace.so             mod_dtrace_prerequest request-start
 9489   http8883     mod_dtrace.so            mod_dtrace_postrequest request-done
 9490   http8883     mod_dtrace.so             mod_dtrace_prerequest request-start
70442   http8885     mod_dtrace.so            mod_dtrace_postrequest request-done
70443   http8885     mod_dtrace.so             mod_dtrace_prerequest request-start
73672   http8887     mod_dtrace.so            mod_dtrace_postrequest request-done
73673   http8887     mod_dtrace.so             mod_dtrace_prerequest request-start
73674   http8888     mod_dtrace.so            mod_dtrace_postrequest request-done
73675   http8888     mod_dtrace.so             mod_dtrace_prerequest request-start
82093   http8889     mod_dtrace.so            mod_dtrace_postrequest request-done
82094   http8889     mod_dtrace.so             mod_dtrace_prerequest request-start
```

Here a single probe definition (in this case `http*:::`) has matched probes across multiple processes (in this case, multiple `httpd` processes). This is one advantage of having a USDT-based provider; the pid provider can also be used to inspect user-land software internals. However, the probes can match only one process at a time (`pid<PID>:::`). This becomes particularly cumbersome for Apache, which can run many `httpd` processes.

The argument types for the `request-start` and `request-done` probes shown previously are as follows:

```
args[0]: conninfo_t *
args[1]: http_reqinfo_t *
```

These are defined in the `/usr/lib/dtrace` translator files. Basic connection information is in `conninfo_t`, which is shared by other providers (nfs, smb, iscsi, ftp):

```
/*
 * The conninfo_t structure should be used by all application protocol
 * providers as the first arguments to indicate some basic information
 * about the connection. This structure may be augmented to accomodate
 * the particularities of additional protocols in the future.
 */
typedef struct conninfo {
      string ci_local;        /* local host address */
      string ci_remote;       /* remote host address */
      string ci_protocol;     /* protocol (ipv4, ipv6, etc) */
} conninfo_t;
```

Information about the HTTP request is in `http_reqinfo_t`:

```
typedef struct {
      string hri_uri;               /* uri requested */
      string hri_user;              /* authenticated user */
      string hri_method;            /* method name (GET, POST, ...) */
      string hri_useragent;         /* "User-agent" header (browser) */
      uint64_t hri_request;         /* request id, unique at a given time */
      uint64_t hri_bytesread;       /* bytes SENT to the client */
      uint64_t hri_byteswritten;    /* bytes RECEIVED from the client */
      uint32_t hri_respcode;        /* response code */
} http_reqinfo_t;
```

This allows powerful one-liners and scripts to be constructed easily. Two examples follow:

Frequency counting the URI component of HTTP requests (the text after `http://hostname`):

```
server# dtrace -n 'http*:::request-start { @[args[1]->hri_uri] = count(); }'
dtrace: description 'http*:::request-start ' matched 11 probes
^C

  /shares/export/javadoc/dtrace/                                    1
  /shares/export/javadoc/dtrace/html/                               1
  /shares/export/javadoc/dtrace/html/fast.html                      2
  /shares/export/javadoc/dtrace/html/JavaDTraceAPI.html                7
  /shares/export/javadoc/dtrace/images/JavaDTraceAPI.gif                9
```

This quickly identifies the most popular files while tracing. In this case, it's `JavaDTraceAPI.gif`.

Frequency counting HTTP response codes:

```
server# dtrace -n 'http*:::request-done { @[args[1]->hri_respcode] = count(); }'
dtrace: description 'http*:::request-done ' matched 9 probes
^C

      403                1
      404                2
      200               39
```

This output caught a few errors, along with many successful HTTP requests.

This section includes three scripts that use this (mod_dtrace plug-in based) http provider and scripts using the syscall provider.

httpclients.d

This script summarizes throughput load from HTTP clients, measured on the server. This information can also be retrieved from the `httpd` access log; however, extra software would need to be executed to summarize the log data. DTrace does the summary on the fly, and this script can be enhanced to include information beyond what is available in the log.

Script

This is a simple script that demonstrates the http provider described earlier:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing... output every 10 seconds, or Ctrl-C.\n");
}

http*:::request-done
{
        @rbytes[args[0]->ci_remote] = sum(args[1]->hri_bytesread);
        @wbytes[args[0]->ci_remote] = sum(args[1]->hri_byteswritten);
}

profile:::tick-10sec,
dtrace:::END
{
        normalize(@rbytes, 1024);
        normalize(@wbytes, 1024);
        printf("\n %-32s %10s %10s\n", "HTTP CLIENT", "FROM(KB)", "TO(KB)");
        printa(" %-32s %@10d %@10d\n", @rbytes, @wbytes);
        trunc(@rbytes);
        trunc(@wbytes);
}
```

Script httpclients.d

Example

While tracing, the 192.168.56.1 client was performing the most HTTP I/O in terms of throughput, downloading 1.2MB during the second ten-second summary.

```
server# httpclients.d
Tracing... output every 10 seconds, or Ctrl-C.

 HTTP CLIENT                            FROM(KB)     TO(KB)
 192.168.56.2                                  0          1
 192.168.56.31                                 2          9
 192.168.56.1                                 26        322

 HTTP CLIENT                            FROM(KB)     TO(KB)
 192.168.56.31                                 2         11
 192.168.56.1                                 51       1222
[...]
```

httperrors.d

This is a similar summary script to `httpclients.d`, using the http provider to summarize client and server HTTP errors.

Script

To match for HTTP errors, line 11 matches when the HTTP response code is between 400 and 600, which covers client and server errors.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7           trace("Tracing HTTP errors... Hit Ctrl-C for report.\n");
8   }
9
10  http*:::request-done
11  /args[1]->hri_respcode >= 400 && args[1]->hri_respcode < 600/
12  {
13          @[args[0]->ci_remote, args[1]->hri_respcode,
14              args[1]->hri_method, args[1]->hri_uri] = count();
15  }
16
17  dtrace:::END
18  {
19          printf("%8s  %-16s %-4s %-6s %s\n", "COUNT", "CLIENT", "CODE",
20              "METHOD", "URI");
21          printa("%@8d  %-16s %-4d %-6s %s\n", @);
22  }
```

Script httperrors.d

Example

To test this script, a nonexistent `/not_there.html` file was requested six times by the 192.168.1.109 client, each returning 404 (file not found). The `private.html` file returned 403 (permission denied).

```
server# httperrors.d
Tracing HTTP errors... Hit Ctrl-C for report.
^C
   COUNT  CLIENT           CODE METHOD URI
       1  192.168.1.109    403  get    /shares/export/fs1/private.html
       6  192.168.1.109    404  get    /not_there.html
```

httpio.d

This is a simple http provider–based script to show the distribution of sent and received bytes for HTTP requests. Larger sent sizes indicate that larger files are being retrieved from the Web server.

Script

This is a simple but powerful script. As with the earlier scripts, the use of `http*` will match HTTP probes from all processes supporting this provider, which allows the entire pool of `httpd` processes to be traced from one script:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        trace("Tracing HTTP... Hit Ctrl-C for report.\n");
}

http*:::request-done
{
         @["received bytes"] = quantize(args[1]->hri_bytesread);
         @["sent bytes"] = quantize(args[1]->hri_byteswritten);
}
```

Script httpio.d

Example

Most of the HTTP requests returned between 1KB and 32KB of data, shown earlier in the "sent bytes" distribution plot. The "received bytes" distribution shows the size of the client requests, which were all between 1KB and 2KB.

```
server# httpio.d
Tracing HTTP... Hit Ctrl-C for report.
^C

  received bytes
           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 23
            2048 |                                         0

  sent bytes
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@                                       1
             512 |@@                                       1
            1024 |@@@@@@@@@                                5
            2048 |@@@@@@@                                  4
            4096 |@@@                                      2
            8192 |@@@@@@@                                  4
           16384 |@@@@@@@@@                                5
           32768 |                                         0
           65536 |@@                                       1
          131072 |                                         0
```

httpdurls.d

Where an http provider is not available, the syscall provider can be used in clever ways to answer similar observability questions. This script is a quick usage tool for Web servers, frequency counting which URLs are being accessed via the HTTP `GET` method.

Script

This script is built upon four assumptions.

- The Web server process name is `httpd` (configurable on line 6).
- The Web server reads client HTTP requests via the `read()` syscall.
- Client HTTP requests as read by http will begin with `GET`.
- No other reads will occur that begin with the letters `GET`.

These assumptions are fairly safe, and they allow the script to be written using the syscall provider alone.[14] This should be a reasonably robust script while these assumptions stand.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   inline string WEB_SERVER_PROCESS_NAME = "httpd";
6
7   dtrace:::BEGIN
8   {
9          printf("Tracing GET from %s processes... Hit Ctrl-C to end.\n",
10              WEB_SERVER_PROCESS_NAME);
11  }
12
13  syscall::read:entry
14  /execname == WEB_SERVER_PROCESS_NAME/
15  {
16          self->buf = arg1;
17  }
18
19  syscall::read:return
20  /self->buf && arg1 > 10/
21  {
22          this->req = (char *)copyin(self->buf, arg1);
23          this->get = strstr(this->req, "GET") != NULL;
24  }
25
26  syscall::read:return
27  /self->buf && this->get/
28  {
```

14. This is a clever trick I wish I had thought of myself; I first saw HTTP processed in this way in scripts written by Bryan Cantrill and Ryan Matteson.

```
29          this->line = strtok(this->req, "\r");
30          this->word0 = this->line != NULL ? strtok(this->line, " ") : "";
31          this->word1 = this->line != NULL ? strtok(NULL, " ") : "";
32          this->word2 = this->line != NULL ? strtok(NULL, " ") : "";
33  }
34
35  syscall::read:return
36  /this->word0 != NULL && this->word1 != NULL && this->word2 != NULL &&
37      this->word0 == "GET"/
38  {
39          @[stringof(this->word2), stringof(this->word1)] = count();
40  }
41
42  syscall::read:return
43  {
44          self->buf = 0;
45  }
46
47  dtrace:::END
48  {
49          printf("  %-10s %-54s %10s\n", "PROTOCOL", "URL", "COUNT");
50          printa("  %-10s %-54s %@10d\n", @);
51  }
```

Script httpdurls.d

Line 20 checks that the returned bytes were longer than ten characters; requests shorter than this are unlikely to be valid HTTP requests.

This script can be enhanced; for example, during read:entry, the file descriptor could be checked in the `fds[]` array to ensure that it is a socket by adding another conditional check in the predicate (line 14):

```
/execname == WEB_SERVER_PROCESS_NAME && fds[arg0].fi_fs == "sockfs"/
```

Example

The `httpdurls.d` script was run on a local Web server for several minutes to see what URLs were being accessed:

```
server# httpdurls.d
Tracing GET from httpd processes... Hit Ctrl-C to end.
^C
  PROTOCOL   URL                                                         COUNT
  HTTP/1.1   /twiki-pub/TWiki/PatternSkin/striped_blue.gif                   1
  HTTP/1.1   /wik                                                            1
  HTTP/1.1   /wiki                                                           1
  HTTP/1.1   /icons/movesm.png                                               2
  HTTP/1.1   /icons/trash.png                                                2
[...]
  HTTP/1.1   /                                                             775
  HTTP/1.1   /wiki/index.php/Main_Page                                     859
  HTTP/1.1   /icons/light_dis.png                                          904
  HTTP/1.1   /wiki/images/analytics_fc.png                                1105
  HTTP/1.1   /twiki-pub/TWiki/TWikiDocGraphics/pdf.gif                    1520
```

continues

```
    HTTP/1.1   /wiki/index.php/Configuration:SAN                           2003
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/colors.css                     2092
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/pattern.js                     2920
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/TWiki_header.gif               3361
    HTTP/1.1   /icons/light_warn.png                                       3464
    HTTP/1.1   /wiki/index.php/User_Interface:CLI                          4004
    HTTP/1.1   /twiki-pub/TWiki/TWikiDocGraphics/else.gif                  5564
    HTTP/1.1   /twiki-bin/view/FishPublic/WebHome                          6687
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/gradient_yellow.jpg            7401
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/bullet-down.gif                8355
    HTTP/1.1   /twiki-pub/TWiki/PatternSkin/gradient_blue.jpg             18475
```

This particular Web server hosts a wiki. The most-accessed file while the `httpdurls.d` script was tracing was `gradient_blue.jpg`. Information like this may be immediately useful; performance of this Web server could be improved by resaving that JPEG at a higher compression level. A typo can also be seen in the output: Someone attempted to load `/wik` instead of `/wiki`.

weblatency.d

Client-side HTTP activity can be interesting to measure, because it can show why some Web sites take longer to load than others. This script shows which Web server hosts were accessed by stripping their name out of the requested URL and also shows the first-byte latency of the HTTP `GET`s.

Script

The `httpdurls.d` script used `strtok()` to process the HTTP `GET` request; however, `strtok()` wasn't available in the first release of Solaris 10 DTrace. Since I wrote `weblatency.d` to run correctly on all Solaris versions, I needed to avoid using `strtok()`. I achieved similar functionality using an unrolled loop of `strlen()` and `dirname()`. Although this script could be shortened (lines 102 through 126), it's included here as an example of unrolled loops and for anyone using early versions of DTrace.

When this script was included in Mac OS X (`/usr/bin/weblatency.d`), DTrace had `strtok()` available, and the script was rewritten to take advantage of it.

```
1    #!/usr/sbin/dtrace -s
[...]
55   #pragma D option quiet
56
57   /* browser's execname */
58   inline string BROWSER = "mozilla-bin";
59
60   /* maximum expected hostname length + "GET http://" */
61   inline int MAX_REQ = 64;
62
63   dtrace:::BEGIN
64   {
```

```
        printf("Tracing... Hit Ctrl-C to end.\n");
}

/*
 * Trace browser request
 *
 * This is achieved by matching writes for the browser's execname that
 * start with "GET", and then timing from the return of the write to
 * the return of the next read in the same thread. Various stateful flags
 * are used: self->fd, self->read.
 *
 * For performance reasons, I'd like to only process writes that follow a
 * connect(), however this approach fails to process keepalives.
 */
syscall::write:entry
/execname == BROWSER/
{
        self->buf = arg1;
        self->fd = arg0 + 1;
        self->nam = "";
}

syscall::write:return
/self->fd/
{
        this->str = (char *)copyin(self->buf, MAX_REQ);
        this->str[4] = '\0';
        self->fd = stringof(this->str) == "GET " ? self->fd : 0;
}

syscall::write:return
/self->fd/
{
        /* fetch browser request */
        this->str = (char *)copyin(self->buf, MAX_REQ);
        this->str[MAX_REQ] = '\0';

        /*
         * This unrolled loop strips down a URL to it's hostname.
         * We ought to use strtok(), but it's not available on Sol 10 3/05,
         * so instead I used dirname(). It's not pretty - it's done so that
         * this works on all Sol 10 versions.
         */
        self->req = stringof(this->str);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->req = dirname(self->req);
        self->nam = strlen(self->req) > 15 ? self->req : self->nam;
        self->nam = basename(self->nam);

        /* start the timer */
```

continues

```
129         start[pid, self->fd - 1] = timestamp;
130         host[pid, self->fd - 1] = self->nam;
131         self->buf = 0;
132         self->fd  = 0;
133         self->req = 0;
134         self->nam = 0;
135 }
136 
137 /* this one wasn't a GET */
138 syscall::write:return
139 /self->buf/
140 {
141         self->buf = 0;
142         self->fd  = 0;
143 }
144 
145 syscall::read:entry
146 /execname == BROWSER && start[pid, arg0]/
147 {
148         self->fd = arg0 + 1;
149 }
150 
151 /*
152  * Record host details
153  */
154 syscall::read:return
155 /self->fd/
156 {
157         /* fetch details */
158         self->host = stringof(host[pid, self->fd - 1]);
159         this->start = start[pid, self->fd - 1];
160 
161         /* save details */
162         @Avg[self->host] = avg((timestamp - this->start)/1000000);
163         @Max[self->host] = max((timestamp - this->start)/1000000);
164         @Num[self->host] = count();
165 
166         /* clear vars */
167         start[pid, self->fd - 1] = 0;
168         host[pid, self->fd - 1] = 0;
169         self->host = 0;
170         self->fd = 0;
171 }
172 
173 /*
174  * Output report
175  */
176 dtrace:::END
177 {
178         printf("%-32s %11s\n", "HOST", "NUM");
179         printa("%-32s %@11d\n", @Num);
180 
181         printf("\n%-32s %11s\n", "HOST", "AVGTIME(ms)");
182         printa("%-32s %@11d\n", @Avg);
183 
184         printf("\n%-32s %11s\n", "HOST", "MAXTIME(ms)");
185         printa("%-32s %@11d\n", @Max);
186 }
```

Script weblatency.d

Change line 58 to match your browser process name. The DTrace team at Apple changed it to match Safari.

Example

Here we run `weblatency.d` while a Mozilla browser loads the *www.planetsolaris.org* Web site. After the Web site was loaded, Ctrl-C was hit to print the following report:

```
client# weblatency.d
Tracing... Hit Ctrl-C to end.
C^
HOST                                      NUM
static.flickr.com                           1
images.pegasosppc.com                       1
www.planetsolaris.org                       5
blogs.sun.com                               7

HOST                              AVGTIME(ms)
static.flickr.com                          65
blogs.sun.com                             285
images.pegasosppc.com                     491
www.planetsolaris.org                     757

HOST                              MAXTIME(ms)
static.flickr.com                          65
images.pegasosppc.com                     491
blogs.sun.com                             962
www.planetsolaris.org                    3689
```

This gives us insight into which hosts were responsible for the time it took to load the Web site. It turns out that requests to *www.planetsolaris.org* were the slowest, with a maximum time of 3.7 seconds (probably the first request, which incurred a DNS lookup).

DNS Scripts

The Domain Name System (DNS) is a hierarchical naming system that maps human-readable host names to IP addresses. DNS is frequently queried as a result of using network software, such as Web browsers, and the time needed for DNS lookups can cause noticeable latency in such software. DTrace allows queries and latency to be analyzed, on the client and server sides.

The scripts that follow demonstrate DNS tracing using the pid provider, because a stable DNS provider is not yet available. Because of this, these scripts were written to match a particular version of software and may need adjustments for them to execute on other versions.

The pid provider is not the only way to monitor DNS activity. DNS requests can be traced at the network level, such as by watching for packets to and from port 53 (DNS). This is made easier if the udp provider is available.

getaddrinfo.d

The `getaddrinfo()` call is the standard POSIX function for retrieving an address from a node name. By tracing it with DTrace, host name lookups via this function can be observed along with the time to return the address. This can be a starting point for investigating DNS performance on the client, because `getaddrinfo()` calls may be satisfied by DNS queries. It's up to the system configuration to decide whether to use DNS when responding to `getaddrinfo()`.

Although the use of `getaddrinfo()` is common, software can be written to resolve hosts via DNS without calling it. For example, the resolver library could be called directly (libresolv), for which the `getaddrinfo.d` script could be modified to trace the function calls used. It's also possible that software could query a DNS server directly with UDP, which would need to be DTraced differently, such as by socket operations or the udp provider, if available.

Script

The script uses the pid provider and so needs to be directed at a process to trace (`dtrace` options `-c` or `-p`). An advantage of tracing the POSIX API is that this script is expected to be more stable than would normally be the case when using the pid provider, which can trace unstable implementation details from user-land software.

The prototype of `getaddrinfo()` has the node to look up in the first argument, which DTrace provides as `arg0`. Since this will be a string in user-land, it needs to be retrieved by DTrace using `copyinstr()`.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
       printf("%-20s  %-12s %s\n", "TIME", "LATENCY(ms)", "HOST");
}

pid$target::getaddrinfo:entry
{
        self->host = copyinstr(arg0);
        self->start = timestamp;
}

pid$target::getaddrinfo:return
/self->start/
{
        this->delta = (timestamp - self->start) / 1000000;
        printf("%-20Y  %-12d %s\n", walltimestamp, this->delta, self->host);
        self->host = 0;
        self->start = 0;
}
```

Script getaddrinfo.d

Example

The `getaddrinfo.d` was used to measure host name lookup time from the `ping` command, as executed on Solaris:

```
client# getaddrinfo.d -c 'ping phobos'
TIME                    LATENCY(ms)   HOST
2010 May 22 06:40:18   218           phobos
no answer from phobos
```

The time to resolve `phobos` was 218 ms, which for this system included the DNS query time. Repeated calls to `getaddrinfo.d` show that the time becomes much quicker:

```
client# getaddrinfo.d -c 'ping phobos'
TIME                    LATENCY(ms)   HOST
2010 May 22 06:55:08   1             phobos
no answer from phobos
```

This is because the Name Service Cache Daemon (`nscd`) is running, which improves the performance of frequently requested lookups by caching the results. While `getaddrinfo()` traced the request, it returned from cache and did not become a DNS query.

dnsgetname.d

The `dnsgetname.d` script monitors DNS lookups from a DNS server. This can be executed at any time without restarting the DNS server or changing its configuration.

Script

This script is written using the pid provider to trace the internal operation of Berkeley Internet Name Daemon (BIND) version 9, the most commonly used DNS server software. Since this traces BIND internals, this script will require updates to match changes in the BIND software.

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option switchrate=10hz

typedef struct dns_name {
        unsigned int                    magic;
        unsigned char *                 ndata;
```

continues

```
9          /* truncated */
10  } dns_name_t;
11
12  pid$target::getname:entry
13  {
14          self->arg0 = arg0;
15  }
16
17  pid$target::getname:return
18  /self->arg0/
19  {
20          this->name = (dns_name_t *)copyin(self->arg0, sizeof (dns_name_t));
21          printf("%s\n", copyinstr((uintptr_t)this->name->ndata));
22          self->arg0 = 0;
23  }
```

Script dnsgetname.d

On this version of BIND, retrieving a text version of the DNS request turned out to be a little difficult. Ideally, there would be a function call containing the request as a `char *` argument, which DTrace would fetch using `copyinstr()`. A function like this does exist, `dns_name_totext()`, but this isn't called for every request.

The `getname()` function is called for every request and has `dns_name_t *` as the first argument. `dns_name_t` is a struct containing a member that points to the name request data, `ndata`. For DTrace to be able to navigate this struct, it is declared on lines 6 to 10 (at least, enough of it for DTrace to find the `ndata` member), and the `-C` option is used on line 1 to allow such declarations in the DTrace script. That struct is later fetched using `copyin()` on line 20.

This approach can be used to fetch other data of interest from the DNS server; however, the script will be come longer and more brittle, as it becomes more tied to implementation details. To keep things simple, only the lookup query is returned in this script.

Example

Here the `dnsgetname.d` script was executed on a 32-bit build of named (BIND), requiring the use of the `-32` option to `dtrace` to trace correctly. Since the name printed contains binary characters (part of the DNS protocol), a Perl one-liner was added to replace these characters with periods, to aid readability:

```
server# dnsgetname.d -32 -p `pgrep named` | perl -ne '$|=1;$_ =~ s/[^:\w ]/./g;
print "$_\n"'
._nfsv4idmapdomain.sf.test.com.
.mars.sp.sf.test.com.
.140.3.168.192.in.addr.arpa.
.phobos.sf.test.com.
.phobos.test.com.
.phobos.
.phobos.sf.test.com.
```

```
._nfsv4idmapdomain.sf.test.com.
.105.1.168.192.in.addr.arpa.
._nfsv4idmapdomain.sf.test.com.
._nfsv4idmapdomain.sf.test.com.
.188.1.168.192.in.addr.arpa.
._nfsv4idmapdomain.sf.test.com.
```

The script prints out DNS queries in real time. The script can be enhanced to include additional information such as query time.

FTP Scripts

The FTP protocol can be traced in a number of places: via a stable ftp provider (if available), via syscalls, at the socket layer, in the TCP/IP stack, and either in the server or client software directly. The ftp provider is currently available only in the Oracle Sun ZFS Storage Appliance where it is used by FTP Analytics; we hope that it is available elsewhere by the time you are reading this book.

If the ftp provider is not available, the pid provider can be used to trace the activity of the user-land FTP server processes. This is demonstrated in this section for ProFTPD[15] on Solaris and for tnftpd[16] on Mac OS X. Use of the pid provider ties the scripts to a particular version of the FTP software and will need adjustments to execute on other versions.

ftpdxfer.d

The `ftpdxfer.d` script uses the ftp provider to trace FTP data transfer operations that occurred while processing FTP commands.

Script

Short and powerful scripts like this demonstrate the value of stable providers:

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-20s %-8s %9s %-5s %-6s %s\n", "CLIENT", "USER", "LAT(us)",
            "DIR", "BYTES", "PATH");
}
```

continues

15. *www.proftpd.org*
16. *http://freshmeat.net/projects/tnftpd*

```

ftp*:::transfer-start
{
        self->start = timestamp;
}

ftp*:::transfer-done
/self->start/
{
        this->delta = (timestamp - self->start) / 1000;
        printf("%-20s %-8s %9d %-5s %-6d %s\n", args[0]->ci_remote,
            args[1]->fti_user, this->delta, args[1]->fti_cmd,
            args[1]->fti_nbytes, args[1]->fti_pathname);
        self->start = 0;
}
```

Script ftpdxfer.d

The `-Z` option is used with DTrace so that the script can begin tracing even if no `ftpd` processes are running yet. Since it uses a USDT provider, it will probe `ftpd` processes as they are instantiated.

Example

A client read a 1MB file `1m` and then wrote another 1MB file called `1m2`. The `ftpdxfer.d` script shows each read and write data transfer that occurred by `ftpd`:

```
server# ftpdxfer.d
CLIENT               USER       LAT(us) DIR   BYTES  PATH
::ffff:192.168.2.51  brendan        205 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan        118 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan        135 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan        116 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan        117 RETR  49152  /export/fs1/1m
[...]
::ffff:192.168.2.51  brendan        111 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan        108 RETR  49152  /export/fs1/1m
::ffff:192.168.2.51  brendan         53 RETR  16384  /export/fs1/1m
::ffff:192.168.2.51  brendan         60 STOR  2896   /export/fs1/1m2
::ffff:192.168.2.51  brendan         26 STOR  4344   /export/fs1/1m2
::ffff:192.168.2.51  brendan         30 STOR  2896   /export/fs1/1m2
::ffff:192.168.2.51  brendan         28 STOR  2896   /export/fs1/1m2
::ffff:192.168.2.51  brendan         24 STOR  3352   /export/fs1/1m2
::ffff:192.168.2.51  brendan         31 STOR  2896   /export/fs1/1m2
::ffff:192.168.2.51  brendan         31 STOR  2896   /export/fs1/1m2
::ffff:192.168.2.51  brendan         30 STOR  4344   /export/fs1/1m2
::ffff:192.168.2.51  brendan         29 STOR  1448   /export/fs1/1m2
```

The client address is printed in the IPv4 encapsulated in IPv6 format. What's interesting is that the reads were 49,152 bytes each (until the end of the file was reached), whereas the writes have variable size.

ftpdfileio.d

This script summarizes bytes read and written over FTP by filename.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

ftp*:::transfer-done
{
        @[args[1]->fti_cmd, args[1]->fti_pathname] = sum(args[1]->fti_nbytes);
}

dtrace:::END
{
        printf("\n%8s %12s  %s\n", "DIR", "BYTES", "PATHNAME");
        printa("%8s %@12d  %s\n", @);
}
```

Script ftpdfileio.d

Example

Testing some known-sized file transfers (one transfer per file) yields the following:

```
server# ftpdfileio.d
Tracing... Hit Ctrl-C to end.
^C

     DIR        BYTES  PATHNAME
    RETR     10485760  /export/fs1/10m
    STOR     10485760  /export/fs1/10m2
    RETR    104857600  /export/fs1/100m
    RETR   1048576000  /export/fs1/1000m
```

This can be a quick way to determine which files are hot and are causing the most FTP data bytes to be transferred.

proftpdcmd.d

This script traces FTP commands processed by `proftpd`, on the FTP server, providing latency details for each executed command. Unlike `ftpdxfer.d`, which traced data transfers, the `proftpdcmd.d` script traces entire FTP commands, which may consist of many data transfers.

Since this uses the pid provider, it needs to be passed a process ID to execute. This means that these pid provider-based FTP scripts must be run on each `ftpd` PID to see their behavior, which may be tricky if the ftpds are short-lived. (This is another advantage of the stable ftp provider: It can examine FTP activity from any process without needing to know the PID beforehand.)

Script

The pid provider was used to examine the execution of the proftpd source (this is ProFTPD Version 1.3.2e): specifically, the `pr_netio_telnet_gets()` function. This was chosen because it returned a pointer to the FTP command string that was read from the client.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-20s %10s  %s\n", "TIME", "LAT(us)", "FTP CMD");
}

/* on this proftpd version, pr_netio_telnet_gets() returns the FTP cmd */
pid$target:proftpd:pr_netio_telnet_gets:return
{
        self->cmd = copyinstr(arg1);
        self->start = timestamp;
}

pid$target:proftpd:pr_netio_telnet_gets:entry
/self->start/
{
        this->delta = (timestamp - self->start) / 1000;
        /* self->cmd already contains "\r\n" */
        printf("%-20Y %10d  %s", walltimestamp, this->delta, self->cmd);
        self->start = 0;
        self->cmd = 0;
}
```

Script proftpdcmd.d

Latency for these FTP commands is calculated in an unusual way. Most latencies in DTrace are calculated as the time from entry to return or from start to done. Here the time is from return to entry.

We do this because, when the `pr_netio_telnet_gets()` returns, it is handing the FTP command string to `proftpd` to process; when it is next called, it is called because the `proftpd` has finished processing that command and is ready to read the next one. So, command processing latency is the time from `pr_netio_telnet_gets()` return to entry. This entry to return time is actually the wait time for the FTP client command, which, for interactive FTP sessions, would span the keystroke latency as a human typed the FTP command.

Example

The `proftpdcmd.d` script is executed just after an FTP connection is established but before authentication. This was executed on Solaris, which has the `pgrep` command[17] available to find the most recent PID matching the string given (easier than `ps -ef | grep | awk` ...):

```
server# proftpdcmd.d -p `pgrep -n proftpd`
TIME                        LAT(us)  FTP CMD
2010 Jan 12 18:39:48         390428  USER brendan
2010 Jan 12 18:39:51        1758793  PASS test123
2010 Jan 12 18:39:51             80  SYST
2010 Jan 12 18:39:51             68  TYPE I
2010 Jan 12 18:39:57            400  CWD export/fs1
2010 Jan 12 18:40:01            181  PORT 192,168,2,51,141,91
2010 Jan 12 18:40:01          98850  RETR 10m
2010 Jan 12 18:40:03            192  PORT 192,168,2,51,226,19
2010 Jan 12 18:40:04         937522  RETR 100m
2010 Jan 12 18:40:06            216  PORT 192,168,2,51,202,212
2010 Jan 12 18:40:15        9592626  RETR 1000m
2010 Jan 12 18:40:18            211  PORT 192,168,2,51,166,0
2010 Jan 12 18:40:18         104173  STOR 10m2
2010 Jan 12 18:40:22            202  PORT 192,168,2,51,145,65
2010 Jan 12 18:40:23         923753  STOR 100m2
2010 Jan 12 18:40:25            212  PORT 192,168,2,51,147,50
2010 Jan 12 18:40:34        9457777  STOR 1000m2
2010 Jan 12 18:40:38            538  MKD newdir
```

To put the script to the test, several "gets" and "puts" were executed by a client, on files of varying sizes. The latency for each transfer corresponds to the size of the file, as shown earlier.

By default, many FTP clients use active sessions to transfer files, which creates data ports for file transfers as shown previously. The following shows passive FTP transfers:

```
server# proftpdcmd.d -p `pgrep -n proftpd`
TIME                        LAT(us)  FTP CMD
2010 Jan 12 18:41:33         389572  USER brendan
2010 Jan 12 18:41:35        1765402  PASS test123
2010 Jan 12 18:41:35            221  SYST
2010 Jan 12 18:41:35             73  TYPE I
2010 Jan 12 18:41:48            399  PASV
2010 Jan 12 18:41:48             93  RETR 10m
2010 Jan 12 18:41:52            393  CWD export/fs1
2010 Jan 12 18:41:55           2082  PASV
2010 Jan 12 18:41:55         102000  RETR 10m
2010 Jan 12 18:42:02            394  PASV
2010 Jan 12 18:42:02         930729  RETR 100m
```

Note that the output examples include the username and password of the client.[18] See the "Security" section for the implications of this.

17. `pgrep` was written by Mike Shapiro, co-inventor of DTrace.

18. No, that's *not* my real password.

tnftpdcmd.d

This is the same script as `proftpdcmd.d`, written for tnftpd on Mac OS X. The design and output is the same; the only difference is the source code function that was probed:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-20s %10s  %s\n", "TIME", "LAT(us)", "FTP CMD");
}

pid$target:ftpd:getline:return
/arg1 && arg1 != 1/
{
        self->line = copyinstr(arg1);
        self->start = timestamp;
}

pid$target:ftpd:getline:entry
/self->start/
{
        this->delta = (timestamp - self->start) / 1000;
        /* self->line already contains "\r\n" */
        printf("%-20Y %10d  %s", walltimestamp, this->delta, self->line);
        self->start = 0;
        self->line = 0;
}
```

Script tnftpdcmd.d

proftpdtime.d

Building on `proftpdcmd.d`, this script prints linear distribution plots for the command times.

Script

`lquantize()` is used on line 22 with a max of 1000 ms and a step of 10 ms. This can be adjusted as desired.

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4   #pragma D option switchrate=10hz
5
6   dtrace:::BEGIN
7   {
8          printf("Tracing... Hit Ctrl-C to end.\n");
9   }
10
```

```
/* on this proftpd version, pr_netio_telnet_gets() returns the FTP cmd */
pid$target:proftpd:pr_netio_telnet_gets:return
{
        self->cmd = copyinstr(arg1);
        self->start = timestamp;
}

pid$target:proftpd:pr_netio_telnet_gets:entry
/self->start/
{
        this->delta = (timestamp - self->start) / 1000000;
        @[self->cmd] = lquantize(this->delta, 0, 1000, 10);
        self->start = 0;
        self->cmd = 0;
}

dtrace:::END
{
        printf("FTP command times (ms):\n");
        printa(@);
}
```

Script proftpdtime.d

Example

Here a 10MB file was fetched many times, and a 100MB file was fetched a few times:

```
server# proftpdtime.d -p `pgrep -n proftpd`
Tracing... Hit Ctrl-C to end.
^C
FTP command times (ms):

  PORT 192,168,2,51,145,88

           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              10 |                                         0
[...]

  RETR 100m

           value  ------------- Distribution ------------- count
             920 |                                         0
             930 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              2
             940 |@@@@@@@@@@@@@                            1
             950 |                                         0

  RETR 10m

           value  ------------- Distribution ------------- count
              80 |                                         0
              90 |@@@@@@                                   4
             100 |@@@@@                                    3
             110 |@@                                       1
             120 |@@@@@@                                   4
             130 |@@@@@@@@                                 5
             140 |@@@@@@                                   4
             150 |@@                                       1
             160 |@@@@@                                    3
             170 |@@                                       1
             180 |                                         0
```

The distribution plots give a good idea of the latency for clients fetching these files. The 10MB file was fetched between 90 ms and 180 ms, while the 100MB file returned more consistently between 930 ms and 950 ms.

proftpdio.d

Apart from digging deeper into `proftpd`, we can also use DTrace to shower higher-level summaries. This tool is based on `iostat` and shows operations per second, throughput, and command latency. It outputs a one-line summary every five seconds; these settings could be customized.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        interval = 5;
        printf("Tracing... Output every %d seconds.\n", interval);
        printf("  FTPD %4s %8s %8s %8s %8s %10s\n",
            "r/s", "w/s", "kr/s", "kw/s", "cmd/s", "cmd_t(ms)");
        tick = interval;
        @readb = sum(0);                /* trigger output */
}

pid$target:proftpd:pr_netio_read:return
/arg1 > 0/
{
        @writes = count();
        @writeb = sum(arg1);
}

pid$target:proftpd:pr_netio_write:entry
/arg2 > 0/
{
        @reads = count();
        @readb = sum(arg2);
}

pid$target:proftpd:pr_netio_telnet_gets:return
{
        @cmds = count();
        self->start = timestamp;
}

pid$target:proftpd:pr_netio_telnet_gets:entry
{
        this->delta = (timestamp - self->start) / 1000000;
        @svct = avg(this->delta);
}

profile:::tick-1sec
/--tick == 0/
{
        normalize(@reads, interval);
```

```
            normalize(@readb, interval * 1024);
            normalize(@writes, interval);
            normalize(@writeb, interval * 1024);
            normalize(@cmds, interval);

            printa("   %@8d %@8d %@8d %@8d %@8d %@10d\n",
                @reads, @writes, @readb, @writeb, @cmds, @svct);

            clear(@reads); clear(@readb); clear(@writes); clear(@writeb);
            clear(@cmds); clear(@svct);
            tick = interval;
}
```

Script proftpdio.d

Example

In this example, an FTP client read (get) a 1GB file and then wrote (put) it back:

```
server# proftpdio.d -p `pgrep -n proftpd`
Tracing... Output every 5 seconds.
  FTPD  r/s      w/s     kr/s     kw/s    cmd/s  cmd_t(ms)
          0        0        0        0        0          0
         43        0     2048        0        0         48
        423        1    19612        0        1          2
       2386        0   114556        0        0          0
       1529        1    72678        0        1       1654
          0     7340        0    57222        0          0
          0    14674        0   108936        0          0
          0     3555        0    38641        0       9460
         14        0        0        0        0          2
          0        0        0        0        0          0
^C
```

The command time average (`cmd_t`) may be confusing: A five-second interval showed an average `cmd_t` of 9.46 seconds. This is because the command began earlier and completed during that five-second interval.

iSCSI Scripts

Figure 7-2 presents an iSCSI functional diagram.

The Internet Small Computer System Interface (iSCSI) protocol provides SCSI disk access over IP. Unlike NFS and CIFS, which provide access to files on a file system, the iSCSI protocol provides block access for reading and writing to disk devices, without knowledge of what those blocks are used for. Clients may install a file system on iSCSI devices, but all the file system processing is performed on the clients.

The upshot of this is that DTrace on the iSCSI server can see which clients, disks, and blocks have I/O but can't see the operation of the file system that is causing that I/O. To see the file system activity, run DTrace on the client (covered in Chapter 5).

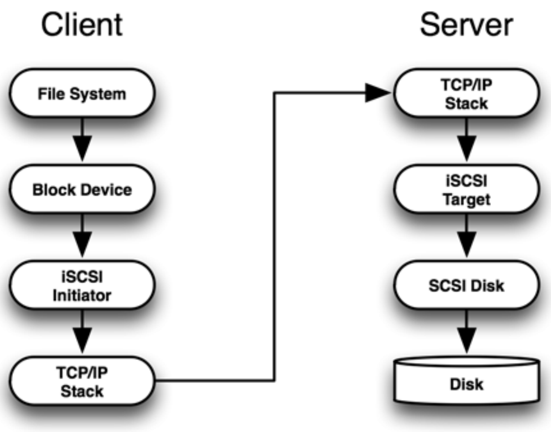


Figure 7-2 iSCSI functional diagram

Providers

The client iSCSI initiator is usually implemented as a kernel disk device driver and can be traced using the io provider for stable I/O events or using the fbt provider to examine kernel internals. The iSCSI target server can be traced with the iscsi provider if it is available in your iSCSI target software version. If it isn't available, your options depend on the iSCSI software used.

The iSCSI target was first implemented as a user-land daemon, `iscsitgtd`, which could be traced using the pid provider. Some versions of `iscsitgtd` came with a USDT provider called *iscsi*.

Later, the COMSTAR[19] project reimplemented the iSCSI target software in the kernel, which can be traced using the fbt provider. Some versions come with an SDT provider also called iscsi, which has a similar interface to the previous iscsi provider for iscsitgtd (the `iscsiwho.d` script that follows works on both). The newer iscsi provider is fully documented in the iscsi provider section of the DTrace Guide.[20] It is currently available in OpenSolaris[21] and Solaris Nevada[22] and is used by most of the scripts in this section. There is also an example of an fbt provider–based script, `iscsiterr.d`.

19. See *http://hub.opensolaris.org/bin/view/Project+comstar/*, which is COMSTAR: Common Multiprotocol SCSI Target.
20. *http://wikis.sun.com/display/DTrace/iscsi+Provider*
21. PSARC 2009/318, CR 6809997, was integrated into Solaris Nevada in May 2009 (snv_116).
22. It is also shipped as part of the Oracle Sun ZFS Storage Appliance, where it powers iSCSI Analytics.

iscsi Provider

Listing iscsi provider probes for COMSTAR iSCSI in Solaris Nevada, circa June 2010, yields the following:

```
solaris# dtrace -ln iscsi:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
14213      iscsi            iscsit                iscsit_op_scsi_cmd xfer-done
14214      iscsi            iscsit                iscsit_op_scsi_cmd xfer-start
14215      iscsi            iscsit                iscsit_op_scsi_cmd scsi-command
14221      iscsi               idm            idm_so_buf_rx_from_ini xfer-start
14222      iscsi               idm              idm_so_buf_tx_to_ini xfer-start
14223      iscsi               idm                   idm_sotx_thread xfer-done
14224      iscsi               idm              idm_so_buf_tx_to_ini xfer-done
14225      iscsi               idm                 idm_so_rx_dataout xfer-done
14226      iscsi               idm             idm_so_free_task_rsrc xfer-done
14230      iscsi               idm                        idm_pdu_rx login-command
14231      iscsi               idm                        idm_pdu_rx logout-command
14232      iscsi               idm            idm_pdu_rx_forward_ffp data-receive
14233      iscsi               idm                        idm_pdu_rx data-receive
14234      iscsi               idm            idm_pdu_rx_forward_ffp task-command
14235      iscsi               idm                        idm_pdu_rx task-command
14236      iscsi               idm            idm_pdu_rx_forward_ffp nop-receive
14237      iscsi               idm                        idm_pdu_rx nop-receive
14238      iscsi               idm            idm_pdu_rx_forward_ffp text-command
14239      iscsi               idm                        idm_pdu_rx text-command
14240      iscsi               idm                        idm_pdu_tx login-response
14241      iscsi               idm                        idm_pdu_tx logout-response
14242      iscsi               idm                        idm_pdu_tx async-send
14243      iscsi               idm                        idm_pdu_tx scsi-response
14244      iscsi               idm                        idm_pdu_tx task-response
14245      iscsi               idm            idm_so_send_buf_region data-send
14246      iscsi               idm                        idm_pdu_tx data-send
14247      iscsi               idm                        idm_pdu_tx data-request
14248      iscsi               idm                        idm_pdu_tx nop-send
14249      iscsi               idm                        idm_pdu_tx text-response
```

The previous listing also highlights the locations of the probes in the kernel, by showing the kernel functions that contain them (FUNCTION column). These can be treated as starting points should it becomes necessary to examine the source code or to trace using the fbt provider.

fbt Provider

To investigate the fbt provider for both the iSCSI target server and the iSCSI client initiator, we conducted a quick experiment. Every related fbt probe was frequency counted while a client performed 1,234 iSCSI reads.

On the iSCSI Target Server. Although the function names will be foreign (unless you have already studied the iSCSI kernel code), this one-liner still serves as a quick way to gauge iSCSI target server activity:

```
server# dtrace -n 'fbt:iscsit::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:idm::entry,fbt:iscsit::entry ' matched 350 probes
^C
[...]
  iscsit_buf_xfer_cb                                            1242
  iscsit_build_hdr                                              1242
  iscsit_dbuf_alloc                                             1242
  iscsit_dbuf_free                                              1242
  iscsit_xfer_scsi_data                                         1242
  iscsit_cmd_window                                             1245
  iscsit_lport_task_free                                        1245
  iscsit_op_scsi_cmd                                            1245
  iscsit_set_cmdsn                                              1245
[...]
```

These probes fired close to the known client read count of `1234`, suggesting that these are related to the processing of iSCSI I/O. The functions `iscsit_op_scsi_cmd()` and `iscsit_xfer_scsi_data()` look like promising places to start tracing, just based on their names. This one-liner traced all the function calls from the `iscsit` module; the `idm` module can also be examined for iSCSI activity.

On the iSCSI Client Initiator. This one-liner shows the probes that fired for the `iscsi` module on the client; it also gives a sense of activity:

```
client# dtrace -n 'fbt:iscsi::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:iscsi::entry ' matched 470 probes
^C
[...]
  iscsi_rx_process_data_rsp                                     1242
  iscsi_cmd_state_active                                        1245
  iscsi_cmd_state_completed                                     1245
  iscsi_cmd_state_free                                          1245
  iscsi_cmd_state_pending                                       1245
  iscsi_dequeue_pending_cmd                                     1245
  iscsi_enqueue_cmd_head                                        1245
  iscsi_enqueue_completed_cmd                                   1245
  iscsi_enqueue_pending_cmd                                     1245
  iscsi_iodone                                                  1245
  iscsi_net_sendmsg                                             1245
  iscsi_net_sendpdu                                             1245
  iscsi_sess_release_itt                                        1245
  iscsi_sess_reserve_itt                                        1245
  iscsi_tran_destroy_pkt                                        1245
  iscsi_tran_init_pkt                                           1245
  iscsi_tran_start                                              1245
  iscsi_tx_cmd                                                  1245
  iscsi_tx_scsi                                                 1245
  iscsi_net_recvdata                                            1247
  iscsi_net_recvhdr                                             1247
  iscsi_rx_process_hdr                                          1247
  iscsi_rx_process_itt_to_icmdp                                 1247
  iscsi_sna_lt                                                  1247
  iscsi_update_flow_control                                     1247
[...]
```

Many probes could be used to trace activity; one in particular is able to trace both read and write I/O on completion: `iscsi_iodone()`. It can be seen in the stack backtrace for the `io:::done` probe, measured here with both read and write I/O:

```
client# dtrace -n 'io:::done { @[stack()] = count(); }'
dtrace: description 'io:::done ' matched 4 probes
^C
              sd`sd_buf_iodone+0x62
              sd`sd_mapblockaddr_iodone+0x48
              sd`sd_return_command+0x158
              sd`sdintr+0x521
              scsi_vhci`vhci_intr+0x688
----->        iscsi`iscsi_iodone+0xc9
              iscsi`iscsi_cmd_state_completed+0x36
              iscsi`iscsi_cmd_state_machine+0xbf
              iscsi`iscsi_ic_thread+0x119
              iscsi`iscsi_threads_entry+0x15
              genunix`taskq_thread+0x1a7
              unix`thread_start+0x8
             4620
```

This is from `uts/common/io/scsi/adapters/iscsi/iscsi_io.c`:

```
void
iscsi_iodone(iscsi_sess_t *isp, iscsi_cmd_t *icmdp)
[...]
```

The `iscsi_sess_t` struct has various members of interest, including the session name:

```
client# dtrace -n 'fbt::iscsi_iodone:entry { trace(stringof(args[0]->sess_name)); }'
dtrace: description 'fbt::iscsi_iodone:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  5  61506          iscsi_iodone:entry iqn.1986-03.com.sun:02:a9877ea7-64d2-ecf4-fe12-
daafa92c015c
  5  61506          iscsi_iodone:entry iqn.1986-03.com.sun:02:a9877ea7-64d2-ecf4-fe12-
daafa92c015c
  0  61506          iscsi_iodone:entry iqn.1986-03.com.sun:02:ea02ce08-d6cb-c810-8540-
b4237b3f8128
  0  61506          iscsi_iodone:entry iqn.1986-03.com.sun:02:32bd3316-53
[...]
```

io Provider

Since the client iSCSI initiator is a disk driver, it can be examined from the io provider. To demonstrate this, the `iosnoop` script from Chapter 4, Disk I/O, was run on a client while it wrote a series of 1MB I/O, beginning at an offset of 100MB:

```
client# iosnoop -se
STIME          DEVICE    UID   PID D    BLOCK   SIZE       COMM PATHNAME
2845748137     sd19        0  1039 R        0    512         dd <none>
2845748471     sd19        0  1039 R        0    512         dd <none>
2845748806     sd19        0  1039 R        0    512         dd <none>
2845751576     sd19        0  1039 W   204800 1048576        dd <none>
2845779437     sd19        0  1039 W   206848 1048576        dd <none>
2845809122     sd19        0  1039 W   208896 1048576        dd <none>
2845838194     sd19        0  1039 W   210944 1048576        dd <none>
2845867094     sd19        0  1039 W   212992 1048576        dd <none>
^C
```

sd19 on the client is the iSCSI device. The block offset for the writes begins with block 204800, because the io provider's block offset is usually given in terms of 512 bytes (204800 x 512 = 100MB). The starting time stamp was printed so that the output could be post sorted if it was shuffled (multi-CPU client).

See Chapter 4 for more examples of the io provider.

iscsiwho.d

The `iscsiwho.d` script summarizes accesses by client IP address and iSCSI event. It is executed on the iSCSI target server.

Script

On line 10, the provider is matched using `iscsi*`. This matches both the `iscsitgtd` and COMSTAR versions of the iscsi provider (which have names `iscsi<PID>` and `iscsi`, respectively). Along with using common arguments, this script will execute on both provider versions:

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7          printf("Tracing iSCSI... Hit Ctrl-C to end.\n");
8   }
9
10  iscsi*:::
11  {
12          @events[args[0]->ci_remote, probename] = count();
13  }
14
15  dtrace:::END
16  {
17          printf("   %-26s %14s %8s\n", "REMOTE IP", "iSCSI EVENT", "COUNT");
18          printa("   %-26s %14s %@8d\n", @events);
19  }
```

Script iscsiwho.d

Example

The `iscsiwho.d` script was run on both versions of the iSCSI target server software, while a client performed a streaming read to an iSCSI device. Here's an example for the `iscsitgtd` iSCSI software:

```
server# iscsiwho.d
Tracing iSCSI... Hit Ctrl-C to end.
^C
   REMOTE IP                    iSCSI EVENT    COUNT
   192.168.100.5                nop-receive        1
   192.168.100.5                   nop-send        1
   192.168.100.7                nop-receive        1
   192.168.100.7                   nop-send        1
   192.168.100.5              scsi-response        3
   192.168.2.30                 nop-receive        3
   192.168.2.30                    nop-send        3
   192.168.2.55                 nop-receive        3
   192.168.2.55                    nop-send        3
   192.168.100.5                  data-send     5315
   192.168.100.5               scsi-command     5318
```

And here's an example for the COMSTAR iSCSI software:

```
server# iscsiwho.d
Tracing iSCSI... Hit Ctrl-C to end.
^C
   REMOTE IP                    iSCSI EVENT    COUNT
   192.168.100.5                nop-receive        1
   192.168.100.5                   nop-send        1
   192.168.2.55                 nop-receive        2
   192.168.2.55                    nop-send        2
   192.168.2.53               scsi-response        7
   192.168.2.53                   data-send     5933
   192.168.2.53                   xfer-done     5933
   192.168.2.53                  xfer-start     5933
   192.168.2.53                scsi-command     5936
```

The clients performed more than 5,000 reads while tracing, shown earlier in the iSCSI event counts. The iSCSI provider probe names are shown in the output of `iscsiwho.d`, so a client read is a `data-send` from the iSCSI server's perspective.

COMSTAR iSCSI added `xfer-start` and `xfer-done` probes because it supports iSER: iSCSI over remote DMA (RDMA). iSER is able to perform faster data transfers by bypassing the usual kernel code paths (using RDMA). Although that's good for performance, it's bad for DTrace observability since the `data-send`/`data-receive` probes do not fire. The `xfer-start`/`xfer-done` probes were added to ensure that these operations have some visibility, because they always fire whether or not iSER is used.

iscsirwsnoop.d

The `iscsirwsnoop.d` script traces iSCSI send and receive probes on the iSCSI target server, printing client details as it occurs.

Script

This version is for the USDT `iscsitgtd` provider:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
       printf("%-16s %-18s %2s %-8s %6s\n", "TIME(us)", "CLIENT", "OP",
           "BYTES", "LUN");
}

iscsi*:::data-send
{
        printf("%-16d %-18s %2s %-8d %6d\n", timestamp / 1000,
            args[0]->ci_remote, "R", args[1]->ii_datalen, args[1]->ii_lun);
}

iscsi*:::data-receive
{
        printf("%-16d %-18s %2s %-8d %6d\n", timestamp / 1000,
            args[0]->ci_remote, "W", args[1]->ii_datalen, args[1]->ii_lun);
}
```

Script iscsirwsnoop.d

This may work on COMSTAR iSCSI (if DTrace complains about insufficient registers, delete the `*`s). However, COMSTAR iSCSI supports RDMA I/O, which bypasses the `data-send` and `data-receive` probes. Because of this, `iscsirwsnoop.d` has been rewritten for COMSTAR iSCSI.

This version is for the SDT kernel iscsi provider:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-16s %-18s %2s %-8s %6s\n", "TIME(us)", "CLIENT", "OP",
            "BYTES", "LUN");
}

iscsi:::xfer-start
{
        printf("%-16d %-18s %2s %-8d %6d\n", timestamp / 1000,
            args[0]->ci_remote, arg8 ? "R" : "W", args[2]->xfer_len,
```

```
                args[1]->ii_lun);
    }
```

Script iscsirwsnoop.d, SDT version

Examples

Here a client created a UFS file system on an iSCSI device. The events were traced on the iSCSI server using `iscsirwsnoop.d`:

```
server# iscsirwsnoop.d
TIME(us)          CLIENT              OP BYTES        LUN
23897801387       192.168.100.5        R 36             0
23897801642       192.168.100.5        R 36             0
23897801991       192.168.100.5        R 512            0
23897802287       192.168.100.5        R 512            0
23897802635       192.168.100.5        R 512            0
23897803137       192.168.100.5        R 36             0
23897803353       192.168.100.5        R 36             0
23897803696       192.168.100.5        R 512            0
23897804033       192.168.100.5        R 512            0
23897804360       192.168.100.5        R 512            0
23897804608       192.168.100.5        R 36             0
23897804830       192.168.100.5        R 36             0
23897805140       192.168.100.5        R 512            0
23897805480       192.168.100.5        R 512            0
23897805826       192.168.100.5        R 512            0
[...]
23900186904       192.168.100.5        R 8192           0
23900186943       192.168.100.5        R 8192           0
23900186972       192.168.100.5        R 8192           0
23900186998       192.168.100.5        R 8192           0
23900187041       192.168.100.5        R 8192           0
23900187075       192.168.100.5        R 8192           0
23900187102       192.168.100.5        R 8192           0
23900187250       192.168.100.5        R 8192           0
[...]
```

The smaller writes early on are likely to be for the ZFS uberblocks, and later the larger writes are likely for the inode tables. See the one-liners to examine the offset of these operations.

The `OP` (operation) column shows what operation the client performed. So, a client read is traced by the iSCSI server probe `data-send`.

iscsirwtime.d

This script traces iSCSI read and write times from the iSCSI target server, printing results as distribution plots and by client and target.

Script

This uses the `xfer-start`/`xfer-done` probes to measure the time of iSCSI data transfers. To calculate transfer time, the starting time stamp is saved in the start associative array keyed on arg1, and arg1 is unique for each data transfer and is used on the xfer-done probe to retrieve the starting time from the start associative array so that the transfer time can be calculated.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

inline int TOP_TARGETS = 10;

dtrace:::BEGIN
{
       printf("Tracing iSCSI target... Hit Ctrl-C to end.\n");
}

iscsi:::xfer-start
{
        start[arg1] = timestamp;
}

iscsi:::xfer-done
/start[arg1] != 0/
{
        this->elapsed = timestamp - start[arg1];
        @rw[arg8 ? "read" : "write"] = quantize(this->elapsed / 1000);
        @host[args[0]->ci_remote] = sum(this->elapsed);
        @targ[args[1]->ii_target] = sum(this->elapsed);
        start[arg1] = 0;
}

dtrace:::END
{
        printf("iSCSI read/write distributions (us):\n");
        printa(@rw);

        printf("\niSCSI read/write by client (total us):\n");
        normalize(@host, 1000);
        printa(@host);

        printf("\niSCSI read/write top %d targets (total us):\n", TOP_TARGETS);
        normalize(@targ, 1000);
        trunc(@targ, TOP_TARGETS);
        printa(@targ);
}
```

Script iscsirwtime.d

Example

While iscsirwtime.d was tracing, a client was performing both small and large reads:

```
server# iscsirwtime.d
Tracing iSCSI target... Hit Ctrl-C to end.
^C
iSCSI read/write distributions (us):
```

```
  read
           value  ------------- Distribution ------------- count
               8 |                                         0
              16 |                                         6
              32 |                                         4
              64 |                                         0
             128 |                                         0
             256 |@@@@@@@@@@                               357
             512 |@                                        43
            1024 |                                         1
            2048 |@@@@@@@@@@@@@@@@@                        591
            4096 |@@@@@@@@@@@                              391
            8192 |                                         0


iSCSI read/write by client (total us):

  192.168.2.53                                              3705661

iSCSI read/write top 10 targets (total us):

  iqn.1986-03.com.sun:02:a9877ea7-64d2-ecf4-fe12-daafa92c015c          1839570
  iqn.1986-03.com.sun:02:32bd3316-538a-ca45-89de-d0ff00d7a2d1          1866091
```

The read distribution shows two groups, one of faster I/O between 256 us and 511 us and one slower between 2 ms and 8 ms. The slower is likely to be for the larger-sized I/O, which can also be identified with DTrace (see the iSCSI one-liners).

iscsicmds.d

Since the iSCSI protocol encapsulates SCSI, it can be interesting to examine the SCSI commands that are being performed. The `iscsicmds.d` script does this, showing SCSI command by client.

Script

This borrows the SCSI command translation table from `scsicmds.d` in Chapter 4, Disk I/O:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

string scsi_cmd[uchar_t];

dtrace:::BEGIN
{
        /*
         * The following was generated from the SCSI_CMDS_KEY_STRINGS
         * definitions in /usr/include/sys/scsi/generic/commands.h using sed.
         */
        scsi_cmd[0x00] = "test_unit_ready";
        scsi_cmd[0x01] = "rezero/rewind";
        scsi_cmd[0x03] = "request_sense";
        scsi_cmd[0x04] = "format";
```

continues

```
        scsi_cmd[0x05] = "read_block_limits";
        scsi_cmd[0x07] = "reassign";
        scsi_cmd[0x08] = "read";
        scsi_cmd[0x0a] = "write";
        scsi_cmd[0x0b] = "seek";
[...see scsicmds.d...]
        scsi_cmd[0xAF] = "verify(12)";
        scsi_cmd[0xb5] = "security_protocol_out";

        printf("Tracing... Hit Ctrl-C to end.\n");
}

iscsi:::scsi-command
{
        this->code = *args[2]->ic_cdb;
        this->cmd = scsi_cmd[this->code] != NULL ?
            scsi_cmd[this->code] : lltostr(this->code);
        @[args[0]->ci_remote, this->cmd] = count();
}

dtrace:::END
{
         printf("  %-24s %-36s  %s\n", "iSCSI CLIENT", "SCSI COMMAND", "COUNT");
         printa("  %-24s %-36s  %@d\n", @);
}
```

Script iscsicmds.d

Example

While tracing, a client created a UFS file system on an iSCSI target:

```
server# iscsicmds.d
Tracing... Hit Ctrl-C to end.
^C
  iSCSI CLIENT             SCSI COMMAND                          COUNT
  192.168.100.4            synchronize_cache                     5
  192.168.2.53             synchronize_cache                     7
  192.168.100.4            test_unit_ready                       9
  192.168.100.4            mode_sense                            15
  192.168.2.53             test_unit_ready                       18
  192.168.2.53             mode_sense                            27
  192.168.100.4            read                                  28
  192.168.100.4            read(10)                              38
  192.168.2.53             read                                  56
  192.168.2.53             read(10)                              79
  192.168.100.4            write(10)                             2138
  192.168.2.53             write(10)                             4277
```

While tracing, client 192.168.2.53 performed 4,277 `write(10)` SCSI operations over iSCSI.

iscsiterr.d

The iscsi provider currently doesn't provide probes for tracing iSCSI errors. This can be extracted from fbt for the COMSTAR iSCSI target software. Because this script is fbt provider–based, for this to keep working, it will need adjustments to match the underlying iSCSI implementation as it changes.

Script

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option switchrate=10hz

typedef enum idm_status {
        IDM_STATUS_SUCCESS = 0,
        IDM_STATUS_FAIL,
        IDM_STATUS_NORESOURCES,
        IDM_STATUS_REJECT,
        IDM_STATUS_IO,
        IDM_STATUS_ABORTED,
        IDM_STATUS_SUSPENDED,
        IDM_STATUS_HEADER_DIGEST,
        IDM_STATUS_DATA_DIGEST,
        IDM_STATUS_PROTOCOL_ERROR,
        IDM_STATUS_LOGIN_FAIL
} idm_status_t;

dtrace:::BEGIN
{
        status[IDM_STATUS_FAIL] = "FAIL";
        status[IDM_STATUS_NORESOURCES] = "NORESOURCES";
        status[IDM_STATUS_REJECT] = "REJECT";
        status[IDM_STATUS_IO] = "IO";
        status[IDM_STATUS_ABORTED] = "ABORTED";
        status[IDM_STATUS_SUSPENDED] = "SUSPENDED";
        status[IDM_STATUS_HEADER_DIGEST] = "HEADER_DIGEST";
        status[IDM_STATUS_DATA_DIGEST] = "DATA_DIGEST";
        status[IDM_STATUS_PROTOCOL_ERROR] = "PROTOCOL_ERROR";
        status[IDM_STATUS_LOGIN_FAIL] = "LOGIN_FAIL";

        printf("%-20s  %-20s %s\n", "TIME", "CLIENT", "ERROR");
}

fbt::idm_pdu_complete:entry
/arg1 != IDM_STATUS_SUCCESS/
{
        this->ic = args[0]->isp_ic;
        this->remote = (this->ic->ic_raddr.ss_family == AF_INET) ?
            inet_ntoa((ipaddr_t *)&((struct sockaddr_in *)&
            this->ic->ic_raddr)->sin_addr) :
            inet_ntoa6(&((struct sockaddr_in6 *)&
            this->ic->ic_raddr)->sin6_addr);

        this->err = status[arg1] != NULL ? status[arg1] : lltostr(arg1);
        printf("%-20Y  %-20s %s\n", walltimestamp, this->remote, this->err);
}
```

Script iscsiterr.d

Example

For this example, a client performed large iSCSI I/O and was then rebooted while I/O was in progress. The iSCSI target server encountered a FAIL error for that client:

```
server# iscsiterr.d
TIME                  CLIENT                ERROR
2010 Jan 15 23:28:22  192.168.100.4         FAIL
```

The script is tracing iSCSI errors. Since iSCSI encapsulates SCSI, examining SCSI errors as well may be of interest. See the `scsireasons.d` script from Chapter 4.

Fibre Channel Scripts

As with iSCSI block I/O, Fibre Channel (FC) block I/O can also be traced on the server and client if DTrace is available. An fc provider exists for FC target tracing, which is fully documented in the fc provider section of the DTrace Guide.[23] It is currently available in OpenSolaris[24] and Solaris Nevada.[25] Listing the fc provider probes on Solaris Nevada, circa June 2010, yields the following:

```
solaris# dtrace -ln fc:::
   ID   PROVIDER          MODULE                          FUNCTION NAME
65315         fc             fct                fct_process_plogi rport-login-end
65316         fc             fct                fct_process_plogi rport-login-start
65317         fc             fct                     fct_do_flogi fabric-login-end
65318         fc             fct                     fct_do_flogi fabric-login-start
65319         fc             fct      fct_handle_local_port_event link-up
65320         fc             fct      fct_handle_local_port_event link-down
78607         fc             fct              fct_handle_rcvd_abts abts-receive
78608         fc             fct              fct_send_scsi_status scsi-response
78609         fc             fct           fct_scsi_data_xfer_done xfer-done
78610         fc             fct                fct_xfer_scsi_data xfer-start
78611         fc             fct                 fct_post_rcvd_cmd scsi-command
78612         fc             fct                   fct_rscn_verify rscn-receive
78613         fc             fct                  fct_process_logo rport-logout-end
78614         fc             fct                  fct_process_logo rport-logout-start
```

If the fc provider is not available, FC may still be traced using the fbt provider, as demonstrated in `fcerror.d`.

23. *http://wikis.sun.com/display/DTrace/fibre+channel+Provider*
24. PSARC 2009/291, CR 6809580, was integrated into Solaris Nevada in May 2009 (snv_115).
25. It is also shipped as part of the Oracle Sun ZFS Storage Appliance, where it powers FC Analytics.

Because FC and iSCSI can be DTraced in similar ways, refer to the "iSCSI Scripts" section for FC script ideas. If the fc provider is available, porting the scripts should be straightforward since the providers have similar interfaces.

fcwho.d

This traces Fibre Channel events on the FC target server and counts which clients and which probe events occurred.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing FC... Hit Ctrl-C to end.\n");
}

fc:::
{
        @events[args[0]->ci_remote, probename] = count();
}

dtrace:::END
{
        printf("   %-26s %14s %8s\n", "REMOTE IP", "FC EVENT", "COUNT");
        printa("   %-26s %14s %@8d\n", @events);
}
```

Script fcwho.d

Example

Here's an example of `fcwho.d` tracing activity from a single client:

```
server# fcwho.d
Tracing FC... Hit Ctrl-C to end.
^C
   REMOTE IP                        FC EVENT    COUNT
   192.168.101.2               scsi-response       11
   192.168.101.2                   xfer-done       22
   192.168.101.2                  xfer-start       22
   192.168.101.2                scsi-command       23
```

fcerror.d

This script traces Fibre Channel packet errors on Solaris Nevada, circa June 2010. It does this using the fbt provider to trace kernel function calls, and so to keep working, it will need adjustments to match the kernel version you are using.

Script

The following code is from `uts/common/io/fibre-channel/impl/fctl.c`:

```
static char *fctl_undefined = "Undefined";
[...]
/*
 * Return number of successful translations.
 *      Anybody with some userland programming experience would have
 *      figured it by now that the return value exactly resembles that
 *      of scanf(3c). This function returns a count of successful
 *      translations. It could range from 0 (no match for state, reason,
 *      action, expln) to 4 (successful matches for all state, reason,
 *      action, expln) and where translation isn't successful into a
 *      friendlier message the relevent field is set to "Undefined"
 */
static int
fctl_pkt_error(fc_packet_t *pkt, char **state, char **reason,
    char **action, char **expln)
{
[...]
        *state = *reason = *action = *expln = fctl_undefined;
[...]
```

Functions like this are a gift in DTrace; information may already be available as translated strings there for the printing. Since this function populates these character pointers, the messages can be printed out only on the return probe:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
       printf("%-20s %-12s %-12s %-12s %-12s\n", "TIME", "STATE", "REASON",
           "ACTION", "EXPLANATION");
}

fbt::fctl_pkt_error:entry
{
        self->state = args[1];
        self->reason = args[2];
        self->action = args[3];
        self->expln = args[4];
}

fbt::fctl_pkt_error:entry
/self->state/
{
        printf("%-20Y %-12s %-12s %-12s %-12s\n", walltimestamp,
            stringof(*self->state), stringof(*self->reason),
            stringof(*self->action), stringof(*self->expln));

        self->state = 0; self->reason = 0; self->action = 0; self->expln = 0;
}
```

Script fcerror.d

NULL checking isn't needed since the strings are set to Undefined by default in the function.

SSH Scripts

SSH is the Secure Shell, an encrypted protocol used for remote shell access, file transfers, and port forwarding. It is typically implemented as a server process called sshd (SSH daemon) and client commands including ssh (Secure Shell) and scp (Secure Copy).

DTrace can be used to examine details of SSH I/O and connections, including the negotiation of encryption algorithms, host key exchanges, and authentication. However, many of these details are already available from OpenSSH, a popular software distribution of SSH, by turning on debug options (for example, ssh -vvv hostname). The scripts that follow show how DTrace can fetch additional information about the activity of the ssh and sshd software. See Chapter 11, Security, for an additional SSH-based script, sshkeysnoop.d.

sshcipher.d

You can use the sshcipher.d script to analyze the CPU cost of encryption ciphers used by SSH. The script can be enhanced to include the CPU cost of compression (if enabled) and other details including the buffer length at encryption time.

Script

This script uses the pid provider to examine the entry and return for any functions containing crypt in their name, in the libcrypto library. Since it uses the pid provider, the script needs to be fed a PID of either ssh or sshd to analyze, via the -p or -c dtrace option.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing PID %d ... Hit Ctrl-C for report.\n", $target);
}

pid$target:libcrypto*:*crypt*:entry
{
         self->crypt_start[probefunc] = vtimestamp;
}

pid$target:libcrypto*:*crypt*:return
/self->crypt_start[probefunc]/
{
```

continues

```
18          this->oncpu = vtimestamp - self->crypt_start[probefunc];
19          @cpu[probefunc, "CPU (ns):"] = quantize(this->oncpu);
20          @totals["encryption (ns)"] = sum(this->oncpu);
21          self->crypt_start[probefunc] = 0;
22  }
23
24  dtrace:::END
25  {
26          printa(@cpu); printa(@totals);
27  }
```

Script sshcipher.d

Some ciphers have `crypt` functions that call `crypt` subfunctions, such as 3DES (`DES_encrypt3()` calls `DES_encrypt2()`). Because of this, the thread-local variable to record the start time is keyed on the function name, on line 12. This ensures that the subfunction calls don't overwrite the start time saved for the parent function.

Example

For the following examples, an `scp` process was executed to copy a large file to a remote host. `scp` runs an `ssh` subprocess to do the encryption, which is traced here.

Default cipher. The following `scp` command line was executed. The cipher algorithm is not specified so that SSH uses the default:

```
client# scp /export/fs1/1g brendan@deimos:/var/tmp
```

The process ID of `ssh` was fetched using the Solaris `pgrep`, and `sshcipher.d` traced it for ten seconds by adding a `dtrace` action at the command line:

```
client# sshcipher.d -p `pgrep -xn ssh` -n 'tick-10sec { exit(0); }'
Tracing PID 3164 ... Hit Ctrl-C for report.

  AES_encrypt                                         CPU (ns):

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@@@@@@@                             220347
            2048 |@                                        10573
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             517175
            8192 |                                         775
           16384 |                                         96
           32768 |                                         0
           65536 |                                         1
          131072 |                                         0


  encryption (ns)                                          3558297733
```

This shows that the default algorithm is AES, for this version of SSH. Since no decrypt functions are shown in the output, either no decryption was performed while tracing, or that function has a name that doesn't contain `crypt` and isn't matched by this script, or the `AES_encrypt()` function is used for both encrypt and decrypt.

The output also showed the CPU time for encryption for each packet, which mostly took between 4 us and 8 us.

Blowfish. The same `scp` command was repeated, this time selecting the Blowfish cipher:

```
client# scp -c blowfish /export/fs1/1g brendan@deimos:/var/tmp
```

Running `sshcpiher.d` yields the following:

```
client# sshcipher.d -p `pgrep -n ssh` -n 'tick-10sec { exit(0); }'
Tracing PID 3145 ... Hit Ctrl-C for report.

  BF_decrypt                                          CPU (ns):
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         3
             512 |@@@@@@@@@@@@@@@@@@@@@@@@                 242
            1024 |@@@@@@@@@@@@@                            132
            2048 |@@                                       23
            4096 |@                                        11
            8192 |                                         0

  BF_encrypt                                          CPU (ns):
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        1136090
             512 |@@@@@@@                                  260700
            1024 |                                         129
            2048 |                                         153
            4096 |                                         282
            8192 |                                         400
           16384 |                                         164
           32768 |                                         1
           65536 |                                         0


  encryption (ns)                                          718476213
```

A key advantage of the Blowfish algorithm is speed, which can be seen by comparing the encryption time of AES to Blowfish, as measured by `sshcipher.d`.

Enhancements

The script can be modified to provide additional information relative to the usage of the cipher interfaces.

Compression. If compression is enabled when using SSH, the CPU overhead can be measured similarly way to encryption by adding the following to the script:

```
pid$target:libz*:inflate:entry,
pid$target:libz*:deflate:entry
{
        self->compress_start = vtimestamp;
}

pid$target:libz*:inflate:return,
pid$target:libz*:deflate:return
/self->compress_start/
{
        this->oncpu = vtimestamp - self->compress_start;
        @cpu[probefunc, "CPU (ns):"] = quantize(this->oncpu);
        @totals["compression (ns)"] = sum(this->oncpu);
        self->compress_start = 0;
}
```

Script addition to sshcipher.d

The additions to the @cpu and @total aggregations will be printed by the existing dtrace:::END action. It may also be desirable to add -Z to the first line so that dtrace can execute even if it can't match the probes. This allows the script to be executed when compression is not used, because SSH may not load the compression library, and so the probes may not be available.

To test this addition, the following scp command was executed:

```
client# scp -C /export/fs1/1g brendan@deimos:/var/tmp
```

Running sshcipher2.d (sshcipher.d plus the previous code) yields the following:

```
client# sshcipher2.d -n 'tick-10sec { exit(0); }' -p `pgrep -xn ssh`
Tracing PID 5395 ... Hit Ctrl-C for report.

  inflate                                          CPU (ns):

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@                         113
            1024 |@@@@@@@@@@@@@@@@@                        117
            2048 |@@@@                                     30
            4096 |@@                                       16
            8192 |                                         0
```

```
  deflate                                    CPU (ns):
           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@@@                 2051
            1024 |                                         8
            2048 |@@@@@@@@                                 670
            4096 |                                         21
            8192 |                                         10
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |@@@@@@@@                                 690
         2097152 |                                         0

  AES_encrypt                                CPU (ns):

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@@@@@@@                             210918
            2048 |@                                        18820
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@              476608
            8192 |                                         1172
           16384 |                                         383
           32768 |                                         1
           65536 |                                         0


  compression (ns)                                          876913379
  encryption (ns)                                          3125429187
```

The totals allow the CPU cost of compression to be compared to encryption for these algorithms. In this case, encryption was about four times more costly.

Cipher Buffer Size. For this version of `ssh`, buffers are encrypted by the function `cipher_crypt()`, which has the following prototype:

```
void
cipher_crypt(CipherContext *cc, u_char *dest, const u_char *src, u_int len)
```

The length of the buffer that is encrypted is the `len` arg, available in DTrace as `arg3`. This can be added to the script to provide details of cipher packet size. The following addition also replaces the `dtrace:::END` action:

```
pid$target:ssh:cipher_crypt:entry
{
        @bytes["cipher average buffer size (bytes)"] = avg(arg3);
        @totals["cipher total (bytes)"] = sum(arg3);
}

```

continues

```
dtrace:::END
{
        printa(@cpu); printa(@bytes); printa(@totals);
}
```

Script addition to sshcipher.d

The summaries at the end of the output now include the average size of each buffer encrypted and the total bytes encrypted. Here we repeat the original AES example:

```
client# sshcipher3.d -n 'tick-10sec { exit(0); }' -p `pgrep -xn ssh`
Tracing PID 5421 ... Hit Ctrl-C for report.

  AES_encrypt                                 CPU (ns):

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@                       345238
            2048 |@                                        17844
            4096 |@@@@@@@@@@@@@@@@@@@@@                    418995
            8192 |                                         822
           16384 |                                         318
           32768 |                                         2
           65536 |                                         3
          131072 |                                         0


  cipher average buffer size (bytes)                          11704

  cipher total (bytes)                                     12546752
  encryption (ns)                                        3016275111
```

The average buffer size is 11KB. This was repeated for Blowfish with the same result.

Now that the script gathers the total bytes encrypted and the total on-CPU encryption time, we can calculate a metric to describe the cipher overhead: CPU nanoseconds per byte, for the different ciphers. Table 7-4 summarizes this result from `scp` tests; this is by no means an authoritative comparison, just a little fun with DTrace.

Table 7-4 Summary of the Results of the scp Tests

Cipher	CPU Nanoseconds/Byte
Blowfish	67
AES	242
3DES	1457

Blowfish wins, in terms of CPU time. The 3DES result was halved to avoid double counting CPU time since it was implemented by `DES_encrypt3()` calling `DES_encrypt2()`, both of which are traced by `sshcipher.d`.

sshdactivity.d

When administrating a system, it can be important to know whether other users are actively using it before certain actions are taken, such as rebooting. Apart from existing operating system tools, which can check the keystroke idle time of logged-in users (such as `w`), the `sshdactivity.d` script shows if any `sshd` processes are actively performing network I/O. This can identify cases where a user has executed a long-running command (for example, a source code build) and is still actively using the system despite their session being considered idle.

Script

The script identifies active SSH sessions by tracing for any `sshd` socket writes and new `sshd` connections:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
       printf("%-20s  %-8s %-8s %-8.8s %s\n", "TIME", "UID", "PID",
            "ACTION", "ARGS");
        my_sshd = $1;
}

syscall::write*:entry
/execname == "sshd" && fds[arg0].fi_fs == "sockfs" && pid != my_sshd/
{
        printf("%-20Y  %-8d %-8d %-8.8s %d bytes\n", walltimestamp, uid, pid,
            probefunc, arg2);
}

syscall::accept*:return
/execname == "sshd"/
{
        printf("%-20Y  %-8d %-8d %-8.8s %s\n", walltimestamp, uid, pid,
            probefunc, "CONNECTION STARTED");
}
```

Script sshdactivity.d

See the One-Liners section for a different method of tracing new `sshd` connections, based on an assumption of `chdir()` behavior.

Example

When first run, the `sshdactivity.d` script may capture activity from itself if it was run over an SSH session (feedback loop):

```
server# sshdactivity.d
TIME                  UID      PID      ACTION   ARGS
2010 May 18 21:53:49  0        3190     write    96 bytes
2010 May 18 21:53:49  0        3190     write    96 bytes
2010 May 18 21:53:49  0        3190     write    96 bytes
2010 May 18 21:53:49  0        3190     write    96 bytes
2010 May 18 21:53:49  0        3190     write    96 bytes
2010 May 18 21:53:49  0        3190     write    96 bytes
. . .
```

This happens because the `dtrace` command prints the header (script line 9), which becomes an `sshd` write, which is traced by the script and printed (line 17). Printing this line causes another `sshd` write, another line to be printed, and so on.

To work around this, an `sshd` PID to ignore can be provided as an optional argument. Having a DTrace script accept optional arguments in this way is possible only when using the `defaultargs` pragma (line 4). The PID to ignore was seen when the script was first run (3190 in the previous output). Adding this yields the following:

```
server# sshdactivity.d 3190
TIME                  UID      PID      ACTION   ARGS
2010 May 18 21:55:04  0        3196     write    176 bytes
2010 May 18 21:55:09  0        3196     write    176 bytes
2010 May 18 21:55:14  0        3196     write    176 bytes
2010 May 18 21:55:14  0        3196     write    112 bytes
2010 May 18 21:55:14  0        3196     write    112 bytes
2010 May 18 21:55:19  0        3196     write    176 bytes
...
```

Now `sshdactivity.d` is only capturing events from other `sshd` processes. Output can be seen every five seconds, which suggests another user is running a tool that updates the screen at this interval (for example, `vmstat 5`). What exactly they are running can be investigated at the command line now that an ancestor PID is known. Solaris can do this easily with the `ptree` command, `ptree PID`, to show all children (and ancestors) from a given PID:

```
server# ptree 3196
1514  /usr/lib/ssh/sshd
  3195  /usr/lib/ssh/sshd
    3196  /usr/lib/ssh/sshd
      3203  -bash
        3225  iostat -xnz 5
```

The user was running `iostat -xnz 5`.

This script can be extended to include other details as desired. A useful addition would be to list the SSH client IP address. One way to do this would be to fetch the host information from the `accept()` syscall, as demonstrated in the `soaccept.d` script in Chapter 6, Network Lower-Level Protocols.

sshconnect.d

When using `ssh` to connect to a remote host, there can be a significant lag between hitting Enter on the `ssh` command and when the password prompt appears. This lag (or latency) can be several seconds long, which for an interactive command can be frustrating for the end user. A number of potential sources of latency could be responsible, in order of execution:

1. `ssh` process initialization time (on-CPU)
2. Reading config files (file system I/O)
3. Target name resolution (typically DNS lookup)
4. TCP connect time (`connect()`)
5. SSH protocol establishment (network I/O)
6. SSH encryption establishment (on-CPU)

The `sshconnect.d` script traces `ssh` commands on the client, providing a summary to identify the source of SSH connection latency.

Script

The aim is to identify which of the six possible causes for latency listed earlier is contributing the most. The following is an example strategy for tracing them using the syscall provider, as implemented by the script. The pid provider could be used instead to examine `ssh` internals. However, such a script will be tied closely to a particular version of the `ssh` software; the syscall based-script is likely to be more robust.

Latencies 1 and 6 are issues of CPU time, which can be examined using vtimestamp deltas. The CPU time between the process starting and calling `connect()` will be measured to answer 1, and the CPU time after `connect()` until the password prompt is printed will be measured to answer 6.

Latencies 2 and 4 can be answered by tracing syscall time using time stamp deltas. Latency 2 includes the syscall time to perform `open()` and `read()` on the config files, and 4 is the syscall time for `connect()` itself.

Latency 3 is also syscall time, although which syscall depends on the library implementation of name resolution (calls such as `getaddrinfo()`). On Solaris, this could be the `doorfs()` syscall, called on the name-service-cache daemon; or it could be `read()` on local files. The most likely syscalls will be traced along with argument information to identify this latency. Since `doorfs()` is probed in the script but doesn't exist on Mac OS X, the `-Z` option is used on line 1 to allow the script to execute on Mac OS X despite listing a nonexistent probe.

Latency 5 is syscall time. This version of `ssh` performs network I/O using a series of syscalls: first to write a request, then to wait for the socket file descriptor to contain the response, and finally to read the response. The syscalls differ between Solaris and Mac OS X:

> **Solaris**: `write()->pollsys()->read()`
>
> **Mac OS X**: `write()->select()->read()`

Most of the network latency occurs during the `pollsys()` or `select()`, which is waiting for the network I/O. There may also be additional `pollsys()->read()` or `select()->read()` iterations to complete the I/O. To identify the network latency from both OSs, the time from either `pollsys()` or `select()` to `read()` completion will be measured.

```
1   #!/usr/sbin/dtrace -Zs
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN { trace("Tracing next ssh connect...\n"); }
6
7   /*
8    * Tracing begins here: ssh process executed
9    */
10  proc:::exec-success
11  /execname == "ssh"/
12  {
13          self->start = timestamp;
14          self->vstart = vtimestamp;
15  }
16  syscall:::entry
17  /self->start/
18  {
19          self->syscall = timestamp;
20          self->arg = "";
21  }
22
23  /*
24   * Include syscall argument details when potentially interesting
25   */
26  syscall::read*:entry,
27  syscall::ioctl*:entry,
28  syscall::door*:entry,
29  syscall::recv*:entry
30  /self->start/
31  {
32          self->arg = fds[arg0].fi_pathname;
33  }
```

```
/*
 * Measure network I/O as pollsys/select->read() time after connect()
 */
syscall::connect:entry
/self->start && !self->socket/
{
        self->socket = arg0;
        self->connect = 1;
        self->vconnect = vtimestamp;
}
syscall::pollsys:entry,
syscall::select:entry
/self->connect/
{
        self->wait = timestamp;
}
syscall::read*:return
/self->wait/
{
        @network = sum(timestamp - self->wait);
        self->wait = 0;
}

syscall:::return
/self->syscall/
{
        @time[probefunc, self->arg] = sum(timestamp - self->syscall);
        self->syscall = 0; self->network = 0; self->arg = 0;
}

/*
 * Tracing ends here: writing of the "Password:" prompt (10 chars)
 */
syscall::write*:entry
/self->connect && arg0 != self->socket && arg2 == 10 &&
    stringof(copyin(arg1, 10)) == "Password: "/
{
        trunc(@time, 5);
        normalize(@time, 1000000);
        normalize(@network, 1000000);
        this->oncpu1 = (self->vconnect - self->vstart) / 1000000;
        this->oncpu2 = (vtimestamp - self->vconnect) / 1000000;
        this->elapsed = (timestamp - self->start) / 1000000;

        printf("\nProcess     : %s\n", curpsinfo->pr_psargs);
        printf("Elapsed     : %d ms\n", this->elapsed);
        printf("on-CPU pre  : %d ms\n", this->oncpu1);
        printf("on-CPU post : %d ms\n", this->oncpu2);
        printa("Network I/O : %@d ms\n", @network);
        printf("\nTop 5 syscall times\n");
        printa("%@8d ms : %s %s\n", @time);

        exit(0);
}

proc:::exit
/self->start/
{
        printf("\nssh process aborted: %s\n", curpsinfo->pr_psargs);
        trunc(@time); trunc(@network); exit(0);
}
```

Script sshconnect.d

The writing of the password prompt is identified via four tests in the predicate.

`self->connect`: This `write()` has occurred after the `connect()`.

`arg0 != self->socket`: This is not a write to the network socket file descriptor.

`arg2 == 10`: This checks the length of the `write()` to see if it is consistent with writing the `Password:` prompt, which is ten characters (includes the space). If this is true, the final test is executed.

`stringof(copyin(arg1, 10)) == "Password: "`: This checks the content of the first ten characters to see whether it matches `"Password: "`. Since this copies the data from user-land to the kernel (`copyin()`), it can be an expensive operation relative to the others and is performed only if all the other tests are positive.

Examples

Examples include host name lookup latency and remote host latency.

Host Name Lookup Latency. When using `ssh` on Solaris to connect to the host `mars.dtrace.com`, it takes about a full second for the `Password:` prompt to appear. Here the `sshconnect.d` script is used to identify the reason for the latency:

```
client# sshconnect.d
Tracing next ssh connect...

Process      : ssh mars.dtrace.com
Elapsed      : 846 ms
on-CPU pre   : 12 ms
on-CPU post  : 54 ms
Network I/O  : 159 ms

Top 5 syscall times
       8 ms : open64
      23 ms : read <unknown>
      29 ms : connect
     159 ms : pollsys
     515 ms : doorfs /var/run/name_service_door
```

The elapsed time is 846 ms, consistent with the experienced latency between running the command and the password prompt. The longest period of latency is identified by the top five syscall listing: `doorfs()` on `/var/run/name_service_door`, taking 515 ms. These door calls are usually issued to perform host name lookups via the Solaris `nscd` (Name Service Cache Daemon), so the longest period of latency is due to resolving `mars.dtrace.com`.

Remote Host Latency. In this example, the ssh connection to host turbot waited almost 20 seconds before printing the password prompt. sshconnect.d was used to identify the latency source:

```
client# sshconnect.d
Tracing next ssh connect...

Process      : ssh root@turbot
Elapsed      : 17523 ms
on-CPU pre   : 11 ms
on-CPU post  : 53 ms
Network I/O  : 17340 ms

Top 5 syscall times
       6 ms : connect
       8 ms : open64
       9 ms : doorfs /var/run/kcfd_door
      14 ms : doorfs <unknown>
   17394 ms : pollsys
```

In this case, the time for the password prompt to appear was 17.5 seconds. This was identified in the output by the "Network I/O" summary, showing 17340 ms. Much of the network I/O latency was encountered during the pollsys() syscall, causing it to show up in the top five syscall times as well. Although this script didn't identify the underlying reason for the latency, it did identify what it isn't: It isn't caused by the client (such as by name services seen in example 1). This means we can focus our analysis on network I/O latency.

Network I/O latency includes the time to route IP packets and the time for the remote SSH daemon (sshd) to respond to the SSH requests. Between these, a latency of 17 seconds is most likely caused by the remote host's sshd, which can be the next target of DTrace analysis.

scpwatcher.d

This short script is an example of clever DTrace scripting (thanks to Bryan Cantrill for the original idea). The problem arises when scp processes have been executed by other users on the system and are taking a long time while consuming network bandwidth. You'd like to know their progress to determine whether to leave them running or to kill them.

Script

There are many ways this script could be written. One may be to use the pid provider to examine scp internals and to dig out progress counters from the scp code. Another may be to attack this from the file system level when scp is sending files and to trace VFS calls to vnodes to determine the progress via the file offset vs. the file size.

The way chosen here is simple: `scp` processes are already writing status information to STDOUT, connected to the other users' terminals; we just can't see it. This DTrace script fetches the STDOUT writes from `scp` processes and reprints it on our screen:

```
1   #!/usr/sbin/dtrace -qs
2
3   inline int stdout = 1;
4
5   syscall::write:entry
6   /execname == "scp" && arg0 == stdout/
7   {
8           printf("%s\n", copyinstr(arg1));
9   }
```

Script scpwatcher.d

Adding a newline on line 8 was a conscious choice to accommodate tracing multiple simultaneous `scp` sessions to differentiate their output lines. It changes the output from updating a single status line to printing a scrolling update.

Example

A file is copied to a remote host. The following is the output from the user's screen for reference:

```
client# scp 100m brendan@192.168.56.1:
Password:
100m             81% |************************************        | 82944 KB   00:02 ETA
```

The status line is frequently updated by `scp`. `scpwatcher.d` traces the writing of this status line for any running `scp` processes on the system:

```
client# scpwatcher.d
100m           0% |                                             |     0      --:-- ETA
100m           8% |***                                          |  8320 KB   00:11 ETA
100m          16% |*******                                      | 16640 KB   00:10 ETA
100m          24% |**********                                   | 24832 KB   00:09 ETA
100m          32% |**************                               | 33408 KB   00:08 ETA
100m          40% |******************                           | 41856 KB   00:07 ETA
100m          48% |*********************                        | 49280 KB   00:06 ETA
100m          56% |*************************                    | 57984 KB   00:05 ETA
100m          64% |****************************                 | 66304 KB   00:04 ETA
100m          72% |********************************             | 74624 KB   00:03 ETA
100m          81% |************************************         | 82944 KB   00:02 ETA
100m          89% |****************************************     | 91520 KB   00:01 ETA
100m          97% |*******************************************  | 99840 KB   00:00 ETA
100m         100% |*********************************************|   100 MB   00:12
```

The output shows what is being written to the user's screen, with newline characters separating each status line update to produce a scrolling output instead of a single line. If desired, `scpwatched.d` could be enhanced to include UID and PID details, as well as the full path of the files being copied.

NIS Scripts

The Network Information Service (NIS) provides centralized configuration and authentication services for network hosts and was originally developed by Sun Microsystems.

nismatch.d

This script traces NIS map lookups (the same as those from the `ypmatch` command).

Script

This simple script demonstrates basic NIS tracing. As with the `dnsgetname.d` script, this uses the pid provider to examine the server software internals, with the trade-off that this script is now tied to a particular version of the NIS server software (`ypserv`).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7          printf("%-20s  %-16s %-16s %s\n", "TIME", "DOMAIN", "MAP", "KEY");
8   }
9
10  pid$target::ypset_current_map:entry
11  {
12          self->map = copyinstr(arg0);
13          self->domain = copyinstr(arg1);
14  }
15
16  pid$target::finddatum:entry
17  /self->map != NULL/
18  {
19          printf("%-20Y  %-16s %-16s %S\n", walltimestamp, self->domain,
20              self->map, copyinstr(arg1));
21  }
```

Script nismatch.d

`finddatum()` takes a `datum` struct as `arg1`; as a shortcut, it's treated as a string pointer on line 20 since the first member is a `char *`. It's printed using `%S` to avoid printing binary characters pulled in by `copyinstr()`; this could be improved by using the length member from the `datum` struct to copy in just the key text.

Example

The `nismatch.d` script was executed on a Solaris NIS server using the `-p` option to match the PID of the NIS server, which was found using the Solaris `pgrep` command. The script traced NIS lookups during an SSH login to an NIS client:

```
server# nismatch.d -p `pgrep ypserv`
TIME                  DOMAIN            MAP              KEY
2010 May 22 20:34:12  newcastle         passwd.byname    YP_SECURE\0
2010 May 22 20:34:12  newcastle         passwd.byname    brendany\b\0
2010 May 22 20:34:12  newcastle         auto.home        YP_SECURE\0
2010 May 22 20:34:12  newcastle         auto.home        brendan\b\b0
2010 May 22 20:34:13  newcastle         passwd.byname    YP_SECURE\0
2010 May 22 20:34:13  newcastle         passwd.byname    YP_MASTER_NAME\0
2010 May 22 20:34:13  newcastle         passwd.byname    YP_SECURE\0
2010 May 22 20:34:13  newcastle         passwd.byname    brendan\b\b\0
2010 May 22 20:34:14  newcastle         passwd.byname    YP_SECURE\0
2010 May 22 20:34:14  newcastle         passwd.byname    brendan\b\b\0
2010 May 22 20:34:15  newcastle         passwd.byuid     YP_SECURE\0
2010 May 22 20:34:15  newcastle         passwd.byuid     138660n\b\b\0
...
```

The role of the `YP_SECURE` key is described in the `ypserv(1M)` man page as a special key to alter the way `ypserv` operates and "causes ypserv to answer only questions coming from clients on reserved ports."

This script could be enhanced to include client details, either via the pid provider to examine more internal functions of `ypserv` or via the syscall provider to examine socket connections (see the socket scripts in Chapter 6, Network Lower-Level Protocols).

LDAP Scripts

The Lightweight Directory Access Protocol (LDAP) providers a hierarchal and secure system of centralized configuration and authentication services for network hosts.

ldapsyslog.d

The `ldapsyslog.d` script shows LDAP requests on an OpenLDAP server, by tracing calls to `syslog()`, even if `syslogd` (the system log daemon) is configured to ignore these messages.

Script

As with `nismatch.d` and `dnsgetname.d`, the internals of the server are examined using the pid provider, with the trade-off that this script is now tied to a particular version of OpenLDAP.

When developing this script, it appeared that a short example would not be possible from the OpenLDAP code. Software is not typically designed for postdebugging with tools such as DTrace, and information in string format suitable for DTrace to fetch and print can sometimes be hard to extract. We found a solution, although it serves more as an example of resourceful tracing than of examining LDAP.

When hunting for strings in code, one trick is to look for any logging functions, which typically write to text-based logs. Logging functions are sometimes written with a test at the top to exit early if logging is not enabled, skipping the actual act of writing to a log file; however, what's important to DTrace is that the function was called regardless, and so the function arguments can be examined whether logging or not is enabled.

OpenLDAP makes `syslog()` calls, which `syslogd` may be ignoring. `syslog()` does take text arguments, but they are variable:

```
void syslog(int priority, const char *message, .../* arguments */);
```

DTrace does not currently have a clean way of dealing with variable argument lists. As a workaround, the `ldapsyslog.d` script waits until the system libraries have converted the variable argument list into a full string and then fetches that string. We found that this could be done by tracing last strlen call while in `syslog()` (there may well be other ways to do this):

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN { printf("Tracing PID %d...\n", $target); }

pid$target::syslog:entry
{
        self->in_syslog = 1;
}

pid$target::strlen:entry
/self->in_syslog/
{
        self->buf = arg0;
}

pid$target::syslog:return
/self->buf/
{
        trace(copyinstr(self->buf));
        self->in_syslog = 0;
        self->buf = 0;
}
```

Script ldapsyslog.d

The result is a script that traces OpenLDAP `syslog()` calls, whether they are logged or not. Since only generic functions were traced, this script may work on other software as well.

Example

This was executed on a Solaris LDAP server running OpenLDAP, while a user logged into SSH on a remote LDAP client. The output shows the LDAP requests:

```
server# ldapsyslog.d -p `pgrep slapd`
Tracing PID 100709...
May 22 23:03:40 slapd[100709]: [ID 848112 FACILITY_AND_PRIORITY] conn=5692 fd=15 ACCEP
T from IP=192.168.2.145:64621 (IP=0.0.0.0:389)
May 22 23:03:40 slapd[100709]: [ID 848112 FACILITY_AND_PRIORITY] conn=5693 fd=15 ACCEP
T from IP=192.168.2.145:43336 (IP=0.0.0.0:389)
May 22 23:03:40 slapd[100709]: [ID 998954 FACILITY_AND_PRIORITY] conn=5692 op=0 SRCH b
ase="ou=people,dc=developers,dc=sf,dc=com" scope=1 deref=3 filter="(&(objectClass=posi
xAccount)(uid=brendan))"
May 22 23:03:40 slapd[100709]: [ID 706578 FACILITY_AND_PRIORITY] conn=5692 op=0 SRCH a
ttr=cn uid uidnumber gidnumber gecos description homedirectory loginshell
May 22 23:03:40 slapd[100709]: [ID 362707 FACILITY_AND_PRIORITY] conn=5692 op=0 SEARCH
 RESULT tag=101 err=32 nentries=0 text=
May 22 23:03:40 slapd[100709]: [ID 338319 FACILITY_AND_PRIORITY] conn=5692 op=1 UNBIND
May 22 23:03:40 slapd[100709]: [ID 952275 FACILITY_AND_PRIORITY] conn=5692 fd=15 close
d
May 22 23:03:40 slapd[100709]: [ID 998954 FACILITY_AND_PRIORITY] conn=5693 op=0 SRCH b
ase="ou=people,dc=developers,dc=sf,dc=com" scope=1 deref=3 filter="(&(objectClass=posi
xAccount)(uid=brendan))"
May 22 23:03:40 slapd[100709]: [ID 706578 FACILITY_AND_PRIORITY] conn=5693 op=0 SRCH a
ttr=cn uid uidnumber gidnumber gecos description homedirectory loginshell
May 22 23:03:40 slapd[100709]: [ID 362707 FACILITY_AND_PRIORITY] conn=5693 op=0 SEARCH
 RESULT tag=101 err=32 nentries=0 text=
May 22 23:03:40 slapd[100709]: [ID 338319 FACILITY_AND_PRIORITY] conn=5693 op=1 UNBIND
May 22 23:03:40 slapd[100709]: [ID 952275 FACILITY_AND_PRIORITY] conn=5693 fd=15 close
d
```

Multiscripts

The scripts in the previous sections demonstrated DTrace for a particular protocol. Since DTrace can observe all layers of the software stack, these protocol scripts can be enhanced by tracing at other layers at the same time. The following script demonstrates this ability.

nfsv3disk.d

This script examines read and write operations at the NFSv3 protocol level and at the disk level, as well as ZFS cache hits. It is written for Oracle Solaris.

Script

Statistics from different providers are printed out on the same line, but to keep this script simple, it doesn't try to associate the activity. This means that the disk

I/O reported may be because of other system activity, not NFSv3. A more complex DTrace script could be written to identify only disk and file system I/O for serving the NFSv3 protocol.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        interval = 5;
        printf("Tracing... Interval %d secs.\n", interval);
        tick = interval;
}

/* NFSv3 read/write */
nfsv3:::op-read-done { @nfsrb = sum(args[2]->res_u.ok.data.data_len); }
nfsv3:::op-write-done { @nfswb = sum(args[2]->res_u.ok.count); }

/* Disk read/write */
io:::done /args[0]->b_flags & B_READ/ { @diskrb = sum(args[0]->b_bcount); }
io:::done /args[0]->b_flags & B_WRITE/ { @diskwb = sum(args[0]->b_bcount); }

/* Filesystem hit rate: ZFS */
sdt:zfs::arc-hit { @fshit = count(); }
sdt:zfs::arc-miss { @fsmiss = count(); }

profile:::tick-1sec
/--tick == 0/
{
        normalize(@nfsrb, 1024 * interval);
        normalize(@nfswb, 1024 * interval);
        normalize(@diskrb, 1024 * interval);
        normalize(@diskwb, 1024 * interval);
        normalize(@fshit, interval);
        normalize(@fsmiss, interval);
        printf("\n   %10s %10s %10s %10s    %10s %10s\n", "NFS kr/s",
            "ZFS hit/s", "ZFS miss/s", "Disk kr/s", "NFS kw/s", "Disk kw/s");
        printa("   %@10d %@10d %@10d %@10d    %@10d %@10d\n", @nfsrb, @fshit,
            @fsmiss, @diskrb, @nfswb, @diskwb);
        trunc(@nfsrb); trunc(@nfswb); trunc(@diskrb); trunc(@diskwb);
        trunc(@fshit); trunc(@fsmiss);
        tick = interval;
}
```

Script nfsv3disk.d

To trace the ZFS hit rate, sdt provider probes are used. Since the sdt provider is not a committed interface, these probes may vanish or change in future versions of the Solaris kernel.

Examples

To test this script, an NFSv3 client performed a streaming disk read of a large file, the first portion of which was cached in DRAM by the ZFS file system on the NFS server:

```
server# nfsv3disk.d
Tracing... Interval 5 secs.

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
        109824       2069          9          0             0          0

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
        109747       1900          0          0             0          0

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
        109900       1898          0          0             0         83

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
        107468       1877          0          0             0          0

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
        102528       1761        209      25446             0          1

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
         97971       1098        770      98227             0         91

      NFS kr/s  ZFS hit/s ZFS miss/s  Disk kr/s      NFS kw/s  Disk kw/s
         96358       1048        758      96705             0          0
^C
```

The first output shows 100 percent read from cache, the last shows 100 percent read from disk. Where the cached portion of the file was exhausted, disk reads begin to occur as well as ZFS misses. Despite disks being much slower than DRAM, the throughput to the application doesn't drop by much, from about 105MB/sec to 95MBs/sec.

The output shows something unexpected: When the file is being entirely read from disk (96705KB/sec from disk, 96358KB/sec over NFS), ZFS hits are still occurring. What's happening can be understood by more DTrace: ZFS identifies this workload as a streaming read and prefetches the file. Sometime later the application requests the data that was previously prefetched and hits from cache. Without prefetch, the throughput is unlikely to have remained so high when the workload hits from disks instead of DRAM.

Summary

This chapter demonstrated tracing of some common application-level protocols, as an extension of the previous chapter on network lower-level protocols. DTrace is able to answer high-level questions, identifying which clients are accessing a server and which files are being accessed, as well as lower-level details as required. Stable providers exist for some of these protocols, making them easy to trace when that provider is available, such as the nfs providers on Solaris Nevada, as demonstrated in this chapter. Tracing when stable providers are not available was also demonstrated for various protocols, by using the unstable fbt and pid providers.

8

Languages

Programmers have a large number of programming languages to choose from when undertaking a software development project, each offering its own unique set of features. Many languages were initially designed to address a specific problem space but over time have evolved to become usable as general-purpose languages. Today's complex application environments are often built using several different languages that each suit specific areas of the application workflow.

The execution environment of the software generated by the programmer generally falls into one of three categories—native code, compiled byte codes, and interpreted code. C and C++ programs are compiled into native machine code that executes directly on the hardware. Some languages are referred to as scripting languages, meaning the code is executed under an interpreter, which handles the compilation and execution of the scripts. Perl and shell are examples of scripting languages. Somewhere in the middle, there are languages that get compiled by a language-specific compiler, not into native code but into byte codes, with the resulting byte codes executed by a virtual machine or byte code interpreter. Java is an example of such a language.

Among the many appealing features of DTrace is that it gives us the capability to use a single tool for analysis, regardless of which language or languages the target application was developed with. The general methodology and use of DTrace is consistent even when observing software written in different languages, although the details, and the actual amount and type of information that can be made available using DTrace, will vary depending on the target language. DTrace was designed to instrument native code, but because the interpreters for scripting and byte code languages execute as native code, DTrace's visibility into the interpreters is a powerful mechanism for observing and analyzing software that runs under an interpreter. Additionally, many popular languages in use today that execute under an interpreter have been enhanced with their own language-specific DTrace provider, greatly enhancing visibility into the software when using DTrace.

DTrace allows programming language execution to be traced, including the execution of function and method calls, object allocation, and, for some languages, line execution. To understand software in intricate detail, the implementation of the programming language can also be studied with DTrace, such as examining when a language interpreter allocates memory. You can answer questions such as the following.

- What functions are being called the most? By which stack trace?
- When is the software calling libc's `malloc()`? For what sizes? And by which stack traces?
- Which functions are returning errors? What were their entry arguments?
- What functions are slow and causing latency?

As an example, the `js_flowinfo.d` script traces the function flow of JavaScript programs, indenting the function name as each is entered:

```
# js_flowinfo.d
  C    PID  DELTA(us)              FILE:LINE TYPE     -- FUNC
  0  11651          2                   .:0    func     -> start
  0  11651         75   func_clock.html:30   func       -> getElementById
  0  11651         51   func_clock.html:-    func       <- getElementById
  0  11651        479   func_clock.html:31   func       -> func_a
  0  11651         25   func_clock.html:21   func         -> getElementById
  0  11651         23   func_clock.html:-    func         <- getElementById
  0  11651      30611   func_clock.html:25   func         -> func_b
  0  11651         79   func_clock.html:13   func           -> getElementById
  0  11651         51   func_clock.html:-    func           <- getElementById
  0  11651      33922   func_clock.html:17   func           -> func_c
  0  11651         75   func_clock.html:6    func             -> getElementById
  0  11651         50   func_clock.html:-    func             <- getElementById
  0  11651      50481   func_clock.html:-    func           <- func_c
  0  11651         24   func_clock.html:-    func         <- func_b
  0  11651         10   func_clock.html:-    func       <- func_a
  0  11651         39   func_clock.html:32   func       -> setTimeout
  0  11651        118   func_clock.html:-    func       <- setTimeout
  0  11651         11   func_clock.html:-    func     <- start
^C
```

Details including delta time, source file, and line number were printed. Should this JavaScript program have a performance issue, a large delta time may be visible that can be immediately associated with a source file and line number. (`js_flowinfo.d` is listed and explained in more detail later in this chapter.)

These are the advantages of using DTrace for tracing JavaScript or any other language.

- Multiple layers of the software stack can be examined together in one tool.
- Observability tools can be customized.
- Extra debugging software need not be installed (where languages are shipped with DTrace providers built in).
- DTrace can examine programs without needing to restart them in debug mode.

Something DTrace cannot do by itself is show source code alongside execution, which some software debuggers and developer environments can do. DTrace could certainly be used by such debuggers to enhance their capabilities, providing insight into other software stack layers.

Developers often get very good at analyzing their layer of the software stack but don't have insight into other layers. For example, DTrace has been used to identify bugs in the operating system library libc, which were encountered by a Java application.

Previous chapters focused mostly on kernel tracing, either via stable providers or tracing the C code using fbt. In this chapter, multiple languages are covered for user-land applications, including numerous examples of using the DTrace provider available for the target language. This is primarily intended for application developers when the language source code is available. Chapter 9, Applications, continues the analysis of software for end users who may not have access to the source code.

Capabilities

DTrace is capable of tracing every layer of the software stack (see Figure 8-1).

Use DTrace to answer the following questions.

1. What functions were called? Why (stack trace)? What were their arguments?
2. What functions errored?
3. What did the function return?
4. What subfunctions did that function call?
5. How long did it take for the function to complete?
6. How long was the function on-CPU?
7. How long was the function waiting off-CPU?

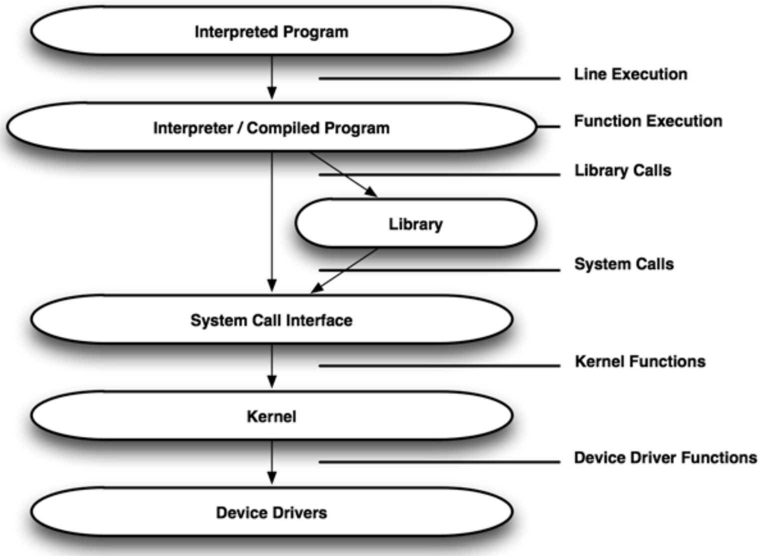


Figure 8-1 Software stack

8. Why did the function/thread leave CPU?
9. What triggered the thread to return to CPU?

Figure 8-2 shows an example of function execution. Two example function returns are illustrated: an early return (2) because of an error (invalid function arguments) and the normal function return (3).

Strategy

To get started using DTrace to examine programming languages, follow these steps (the target of each step is in bold):

Try the DTrace **one-liners** and **scripts** listed in the sections that follow.

In addition to those DTrace tools, familiarize yourself with existing **language debuggers** and **language profilers**, such as Oracle Solaris Studio 12.[1] These are worth fully exploring, because they have been custom-built for

1. See Gove (2007).

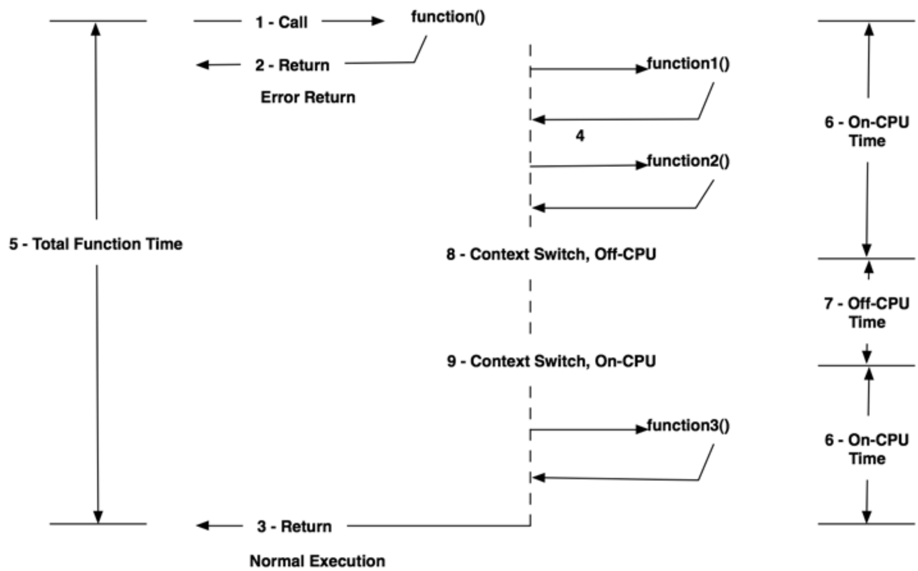


Figure 8-2 Program execution flow

analyzing the target language. The metrics that these retrieve can also show what types of information may be useful to then investigate further with DTrace.

In the target programming language, write tools to generate **known workloads**, such as performing a function a known number of times or with expected high latency. It is *extremely* helpful to have known programs to check your debuggers against. Samples are provided in the "Scripts" section for each language.

Customize and write your own one-liners and scripts that use **specific language providers** (for example, the perl provider), referring to the documentation in the "Providers" section.

To dig deeper than specific providers allow, familiarize yourself with how the software operates by examining **stack backtraces** (see the "One-Liners" section) from various events, including system calls for I/O.

Examine software internals using the **pid provider** and referring to source code if available. For all languages (with the exception of C and C++), this is expected to be difficult, requiring familiarity with the software implementation of the language.

Checklist

Table 8-1 suggests different types of issues that can be examined using DTrace. This can also serve as a checklist to ensure that all obvious types of issues are considered.

Table 8-1 Languages Checklist

Issue	Description
On-CPU time	Functions may be using CPU resources and taking time to complete because of long and complex code paths. This can be identified with DTrace by sampling user stack traces with the profile provider and by measuring the vtimestamp delta between function entry and return.
Off-CPU time	Functions may be taking a long time to complete because of I/O wait time or lock contention. Long latencies cause performance issues and can be identified using DTrace to measure the time stamp delta from function entry to return.
Volume	Function execution can be counted, which can identify whether a call is being made too frequently.
Locks	Waiting on locks can occur both on-CPU (spin) and off-CPU (wait). Locks are used for synchronization of multithreaded applications and, when poorly used, can cause application latency and thread serialization. Use DTrace to examine lock usage by user stack trace.
Memory allocation	Memory allocation via the standard system libraries (`malloc()`, and so on) can be examined using the pid provider, along with entry and return arguments, and user stack trace to explain the code path to the event. Languages may implement their own layer of memory allocation, the workings of which can also be traced using DTrace (for example, Java garbage collection).
Errors	Error state can be examined, whether it is passed as a return value from functions or members of a more complex struct (for example, the io provider `args[0]->b_error`). Errors can be examined from any layer: the application, libraries, system calls, and within the kernel.

Providers

Table 8-2 shows providers of interest to trace programming languages.

If a language you are interested in is not listed here, check whether a provider has been developed since this book was written. New providers are written over time, and existing providers are sometimes enhanced.

If the provider you are interested in is listed here but is not available on your software version, you can try upgrading to the DTrace-enabled version. The language sections in this chapter show the software versions that introduced DTrace providers.

For most of these languages, there are other options if the specific language provider is unavailable: If the source code is open, you can consider writing your own provider (see Appendix E for an example of writing a USDT provider). Or, you can try using the pid provider to trace the internals of the language software. For example, the pid provider can be used to trace the internal operation of `/usr/bin/perl` and `libperl`, providing insight into the operation of Perl programs that are being executed. Although possible, such an approach typically requires familiarity with the software source code and is not recommended unless you already have such familiarity or are prepared to spend significant time gaining it. Because of their complexity, understanding the source code implementation of languages such as Perl is also an advanced programming task. Also note that the pid provider is considered an "unstable" interface because it instruments a specific

Table 8-2 Programming Language Providers

Provider	Description
javascript	JavaScript provider
profile	Samples which functions or stacks are on-CPU
PHP	PHP provider
pid	Traces C and C++ functions and instructions
perl	Perl provider
python	Python provider
hotspot	Java HotSpot VM provider
hotspot_jni	Java HotSpot JNI provider
ruby	Ruby provider
sched	Trace when functions or stacks switch CPU
sh	Bourne shell provider
tcl	Tcl provider

software version, meaning that scripts written that use pid are likely to need updating to work on new versions of the Perl software. See the "pid Provider" section in Chapter 9 for further discussion.

The pid provider can also be used to extend the language provider by examining the operation of the language software in the context of the program being executed, for example, to see when the libc `malloc()` function is called during the execution of programs.

Languages

The sections that follow demonstrate tracing of these languages (in alphabetical order):

Assembly

C

C++

Java

JavaScript

Perl

PHP

Python

Ruby

Shell

Tcl

There is a focus on the language provider for each language (if one exists; see Table 8-2). All languages can be examined using DTrace without a language provider by treating the execution of the program, whether it is a language interpreter or compiled code, like any other application. See Chapter 9, which has a case study that includes JavaScript execution tracing without the JavaScript provider.

The sections that follow show how to retrieve context of these languages within DTrace: the functions and methods being called, from which source files, and other related events such as allocation and garbage collection (if relevant). The rest of the operating system can then be traced in this context by enhancing these scripts with additional probes.

A little extra attention is given to the "Perl" section so that additional script ideas could be demonstrated (the "See Also" scripts). The other sections refer to the

Perl additions as a source of ideas; these additional scripts have been rewritten for many of the other languages and are available in the DTraceToolkit.

To trace any given language, familiarity with that language and its operation is assumed. Numerous books cover each of these languages that can be used for reference.

Assembly

The pid provider is used in various places in this book to trace function execution via the entry and return probes. It also supports instruction offset probes, which allow the tracing of individual instructions in between entry and return. This is tracing at the assembly language level and is possible only for user-land software with the pid provider.

Examining instruction execution is usually only of interest to software developers when debugging code and often only then for particularly nasty bugs that need step-by-step analysis at the instruction level. That this is possible with DTrace is interesting and worth noting, but you can expect to use it rarely. It could be used, for example, to check code path for a function in production, where a debugger cannot be attached to the production code.

The probe specification for instruction tracing is `pid$target:`***`module:function:offset`***, where `$target` can either be a literal process ID or be specified via either the `-p PID` or `-c command` `dtrace(1M)` command-line option. The offset is in hexadecimal.

The contents of registers can be examined via the `uregs[]` array, as documented in the "`uregs[]` Array" section of the "User Process Tracing" chapter of the DTrace Guide.[2] For example, the variable `uregs[R_EAX]` is the `%eax` register on x86.

Example: x86

To demonstrate instruction tracing, the following shows an `mdb(1)` dissassembly of the `strcpy()` function on an x86 server running Oracle Solaris:

```
> strcpy::dis
libc_hwcap2.so.1`strcpy:        pushl  %edi
libc_hwcap2.so.1`strcpy+1:      movl   0xc(%esp),%ecx
libc_hwcap2.so.1`strcpy+5:      movl   0x8(%esp),%edi
libc_hwcap2.so.1`strcpy+9:      movl   %ecx,%eax
libc_hwcap2.so.1`strcpy+0xb:    subl   %edi,%ecx
libc_hwcap2.so.1`strcpy+0xd:    andl   $0x3,%eax
libc_hwcap2.so.1`strcpy+0x10:   je     +0x17    <libc_hwcap2.so.1`strcpy+0x29>
```

2. *http://wikis.sun.com/display/DTrace/User+Process+Tracing*

```
libc_hwcap2.so.1`strcpy+0x12:   subl   $0x4,%eax
libc_hwcap2.so.1`strcpy+0x15:   movb   (%edi,%ecx),%dl
libc_hwcap2.so.1`strcpy+0x18:   movb   %dl,(%edi)
libc_hwcap2.so.1`strcpy+0x1a:   incl   %edi
libc_hwcap2.so.1`strcpy+0x1b:   testb  %dl,%dl
libc_hwcap2.so.1`strcpy+0x1d:   je     +0x3b    <libc_hwcap2.so.1`strcpy+0x5a>
libc_hwcap2.so.1`strcpy+0x1f:   incl   %eax
libc_hwcap2.so.1`strcpy+0x20:   jne    -0xd     <libc_hwcap2.so.1`strcpy+0x15>
libc_hwcap2.so.1`strcpy+0x22:   jmp    +0x5     <libc_hwcap2.so.1`strcpy+0x29>
libc_hwcap2.so.1`strcpy+0x24:   movl   %eax,(%edi)
libc_hwcap2.so.1`strcpy+0x26:   addl   $0x4,%edi
libc_hwcap2.so.1`strcpy+0x29:   movl   (%edi,%ecx),%eax
libc_hwcap2.so.1`strcpy+0x2c:   leal   0xfefefeff(%eax),%edx
[...output truncated...]
```

The instruction offsets can be seen in the `mdb` output after the `+`. All offsets can be traced by leaving the probe name field blank (wildcard), here during execution of `date(1)`:

```
# dtrace -n 'pid$target:libc:strcpy:' -c date
dtrace: description 'pid$target:libc:strcpy:' matched 41 probes
Mon Jul 12 01:51:39 UTC 2010
dtrace: pid 928 has exited
CPU     ID                    FUNCTION:NAME
  8   1414                     strcpy:entry
  8   1415                         strcpy:0
  8   1416                         strcpy:1
  8   1417                         strcpy:5
  8   1418                         strcpy:9
  8   1419                         strcpy:b
  8   1420                         strcpy:d
  8   1421                        strcpy:10       <--- jump from 0x10 to 0x29
  8   1433                        strcpy:29
  8   1434                        strcpy:2c
[...output truncated...]
```

A jump in instruction offset has been labeled in the previous code. By inspecting the dissassembly, this can be identified as a `je` instructions (jump if equal). The source code explains:

```
usr/src/lib/libc/i386/gen/strcpy.s:
[...]
            ENTRY(strcpy)
            push    %edi                        / save reg as per calling cvntn
            mov     12(%esp), %ecx              / src ptr
            mov     8(%esp), %edi               / dst ptr
            mov     %ecx, %eax                  / src
            sub     %edi, %ecx                  / src - dst
            and     $3, %eax                    / check src alignment
            jz      load
            sub     $4, %eax
[...]
```

The assembly is available in this form (with comments!) since these string functions are written by hand and not autogenerated by a compiler (although a compiler has changed it slightly; the `jz` had become a `je`).

To demonstrate tracing an individual instruction and register, the 0xd instruction is traced, and the `%eax` register is fetched using the `uregs[]` array. A bitwise-AND with 3 is applied to match the previous source, showing whether the address is aligned:

```
# dtrace -n 'pid$target:libc:strcpy:d { @[uregs[R_EAX] & 3] = count(); }' -c date
dtrace: description 'pid$target:libc:strcpy:d ' matched 1 probe
Mon Jul 12 02:11:22 UTC 2010
dtrace: pid 944 has exited

                2                2
                0                5
```

This shows that five of the seven calls were aligned. Unaligned addresses execute extra instructions, so identifying them may be of interest for performance analysis.

Finally, the string referenced by `%eax` can be retrieved using `copyinstr()`:

```
# dtrace -n 'pid$target:libc:strcpy:d { trace(copyinstr(uregs[R_EAX])); }' -c date
dtrace: description 'pid$target:libc:strcpy:d ' matched 1 probe
Mon Jul 12 02:23:56 UTC 2010
dtrace: pid 948 has exited
CPU     ID                    FUNCTION:NAME
  8   1413                         strcpy:d   SUNW_OST_OSCMD
  8   1413                         strcpy:d   SUNW_OST_OSCMD
  8   1413                         strcpy:d   UTC
  8   1413                         strcpy:d   UTC
  8   1413                         strcpy:d   UTC
  8   1413                         strcpy:d   Mon
  8   1413                         strcpy:d   Jul
```

C

The C programming language is popular and widely used for writing native code. Its power, flexibility, and relatively simple syntax, along with the broad availability of compilers and debuggers, have made C the language of choice for software development for many years. Several newer languages are based on C at some level, notably C++ and Objective-C.

C code can be traced using the pid provider for user-land software applications and the fbt provider for the kernel (which is mostly written in C). These require certain symbol information to still be present in the binary executable so that DTrace can determine which addresses to dynamically trace.

DTrace allows the entry and return of C functions to be traced and can examine their arguments and return values. Complex arguments such as pointers to structures can be navigated in the same way as other C language constructs, as demonstrated in this section.

This section summarizes tracing C code with DTrace and in particular will explain the difference between tracing user-land C and kernel C. Throughout the book there are many examples of tracing C code, although they are not described as specific C examples. Look for any examples that use the pid or fbt provider. A table of these appears at the end of this section (Table 8-4).

User-Land C

The term *user-land* refers to the address space for software executed by users on the system. This is any software that runs with a process ID, which is where the name of the provider comes from. Listing pid provider probes for an example user-land program, `date(1)`:

```
# dtrace -ln 'pid$target:::entry,pid$target:::return' -c date
   ID   PROVIDER            MODULE                          FUNCTION NAME
96091  pid22793              date                             _start entry
96092  pid22793              date                              __fsr entry
96093  pid22793              date                               main entry
96094  pid22793              date                            setdate entry
96095  pid22793              date                            get_adj entry
96096  pid22793        LM1`ld.so.1                          avl_walk entry
[...6797 lines truncated...]
102893  pid22793         libc.so.1              coll_conv_input_real return
102894  pid22793         libc.so.1                       __strxfrm_sb return
102895  pid22793         libc.so.1                coll_str2weight_sb return
102896  pid22793         libc.so.1                coll_chr2weight_sb return
```

The provider name is `pid` followed by the process ID: Here it was `pid22793` for the executed `date(1)` command. The module field shows the address space object for the functions: The first five show `date` as the module name (which is the `a.out` segment); the last shown are from the libc library.[3]

3. For an understanding of these segments, see the "Linker and Libraries Guide" from the Oracle Solaris Developer Manual collection.

Kernel C

This example lists fbt provider probes for an example kernel module, ZFS:

```
# dtrace -ln fbt:zfs::
   ID   PROVIDER            MODULE                          FUNCTION NAME
44368        fbt               zfs                          buf_hash entry
44369        fbt               zfs                          buf_hash return
44370        fbt               zfs              buf_discard_identity entry
44371        fbt               zfs              buf_discard_identity return
44372        fbt               zfs                     buf_hash_find entry
44373        fbt               zfs                     buf_hash_find return
[...4531 lines truncated...]
48904        fbt               zfs                      sa_set_userp entry
48905        fbt               zfs                      sa_set_userp return
48906        fbt               zfs         zfs_ereport_free_checksum entry
48907        fbt               zfs         zfs_ereport_free_checksum return
```

The ability to trace kernel functions is sometimes used as an introduction to the power of DTrace. The following counts these probes using `wc(1)` on Oracle Solaris:

```
solaris# dtrace -ln fbt::: | wc -l
   70139
```

This shows 70,138 available probes (subtracting the header line), which will be for 35069 kernel functions (one probe for function entry, one for return).

Probes and Arguments

Table 8-3 presents C probes and arguments.

Table 8-3 C Probes and Arguments

Description	Probe	Arguments
User function entry	`pid$target:segment:function:entry`	`arg0..argN: function arguments`
User function return	`pid$target:segment:function:entry`	`arg0: return offset, arg1: return value`
Kernel function entry	`fbt:module:function:entry`	`arg0..argN: function arguments`
Kernel function return	`fbt:module:function:return`	`arg0: return offset, arg1: return value`

The arguments `arg0..argN` are of `uint64_t`. For kernel functions on Oracle Solaris, they may also be available as `args[0..N]`, which are cast to match the correct type.

Struct Types

For kernel tracing, C struct types may already be known to DTrace, allowing immediate navigation of struct members. In Oracle Solaris, this is possible through a facility called *Compact C Type Format* (CTF), which builds type information into the kernel for debuggers to read. Other operating systems have similar facilities that DTrace uses to understand kernel types.

For example, the `zfs_read()` function has a `vnode_t` pointer as the first argument, available in DTrace as `args[0]`. See how this one-liner retrieves the `v_path` member from the struct by simply dereferencing it (then `stringof()` turns the `char` pointer into the string):

```
solaris# dtrace -n 'fbt::zfs_read:entry { @[stringof(args[0]->v_path)] = count(); }'
dtrace: description 'fbt::zfs_read:entry ' matched 1 probe
^C

  /lib/ld.so.1                                                     4
  /usr/bin/ls                                                      5
  /etc/group                                                       6
  /etc/security/policy.conf                                        6
  /etc/passwd                                                     14
  /etc/svc/repository.db                                          47
```

As a more complex example, the `scsicmds.d` script from Chapter 4, Disk I/O, retrieves the device nodename from deep within kernel structures:

```
94 fbt::scsi_transport:entry
95 {
96       this->dev = (struct dev_info *)args[0]->pkt_address.a_hba_tran->tran_hba_dip;
97       this->nodename = this->dev != NULL ?
98           stringof(this->dev->devi_node_name) : "<unknown>";
```

The argument to `scsi_transport()` is a `struct scsi_pkt` pointer, which is walked on line 96 to retrieve a `tran_hba_dip` member, which is then recast as a `struct dev_info` pointer and then walked. All of these types are already known to DTrace, allowing the script to navigate structures in the same way as the kernel code it is tracing.

There are other examples of structure navigation in the `/usr/lib/dtrace` translators. For example, `/usr/lib/dtrace/io.d` translates the mountpoint from `struct buf` using (from Oracle Solaris):

```
translator fileinfo_t < struct buf *B > {
[...]
        fi_mount = B->b_file == NULL ? "<none>" :
            B->b_file->v_vfsp->vfs_vnodecovered == NULL ? "/" :
            B->b_file->v_vfsp->vfs_vnodecovered->v_path == NULL ? "<unknown>" :
            cleanpath(B->b_file->v_vfsp->vfs_vnodecovered->v_path);
```

For user-land tracing of arbitrary software, there may be no built-in structure information for DTrace to use, so struct types must be declared before they can be used. The `-C` option to DTrace executes the preprocessor, allowing types to be defined and header files included in the same way as C so that structs can be navigated. Each dereference requires `copyin()` statements to bring data into the kernel where DTrace is executing.

Includes and the Preprocessor

Header files can be included in D programs using the `-C` option for the preprocessor. An example of this can be seen in the `mmap.d` script from Chapter 5, File Systems, which includes the C header file `sys/mman.h` so that `mmap()` flag definitions can be used in the script.

An example that includes more preprocessor directives is in `kstat_types.d` from the DTraceToolkit, which has the following:

```
 1      #!/usr/sbin/dtrace -Cs
[...]
41      #include <sys/isa_defs.h>
[...]
50
51      fbt::read_kstat_data:entry
52      {
53      #ifdef _MULTI_DATAMODEL
54              self->uk = (kstat32_t *)copyin((uintptr_t)arg1, sizeof (kstat32_t));
55      #else
56              self->uk = (kstat_t *)copyin((uintptr_t)arg1, sizeof (kstat_t));
57      #endif
58              printf("%-16s %-16s %-6s %s:%d:%s\n", execname,
59                  self->uk->ks_class == "" ? "." : self->uk->ks_class,
[...]
```

Line 1 has the `-C` option, allowing line 41 to include the C header file `sys/isa_defs.h`, which has the definition of `_MULTI_DATAMODEL` used by the preprocessor on line 53.

C One-Liners

Here we present several one-liners that provide a solid starting point for observing your executing C programs.

pid Provider

These are mostly demonstrated on the libc library on Oracle Solaris; you can modify the examples as needed. Also, substitute `-p PID` with `-c command` to execute a new command rather than attaching to an already running process.

Count function calls from a segment (for example, libc):

```
dtrace -n 'pid$target:libc::entry { @[probefunc] = count(); }' -p PID
```

Trace a specific function (for example, `fopen()`):

```
dtrace -n 'pid$target:libc:fopen:entry' -p PID
```

Trace function entry arguments (for example, `fdopen()`):

```
dtrace -n 'pid$target:libc:fdopen:entry { trace(arg0); }' -p PID
```

Trace function return value (for example, `fclose()`):

```
dtrace -n 'pid$target:libc:fclose:return { trace(arg1); }' -p PID
```

Trace segment functions with flow indent (for example, `a.out`):

```
dtrace -Fn 'pid$target:a.out::entry,pid$target:a.out::return' -p PID
```

Trace single-function instructions (for example, `strlen()`):

```
dtrace -n 'pid$target:libc:strlen:' -p PID
```

Show user stack trace on function call (for example, `fopen()`):

```
dtrace -n 'pid$target:libc:fopen:entry { ustack(); }' -p PID
```

Count user stack traces for a function call (for example, `fopen()`):

```
dtrace -n 'pid$target:libc:fopen:entry { @[ustack()] = count(); }' -p PID
```

fbt Provider

Count kernel module function calls (for example, `zfs` on Oracle Solaris):

```
dtrace -n 'fbt:zfs::entry { @[probefunc] = count(); }'
```

Count kernel function calls beginning with... (for example, `hfs_`):

```
dtrace -n 'fbt::hfs_*:entry { @[probefunc] = count(); }'
```

Trace a specific kernel function (for example, `arc_read()`):

```
dtrace -n 'fbt::arc_read:entry'
```

Trace kernel function entry arguments (for example, the `zfs_open()` filename):

```
dtrace -n 'fbt::zfs_open:entry { trace(stringof(arg0)); }'
```

Trace kernel function return value (for example, `zfs_read()`):

```
dtrace -n 'fbt::zfs_read:return { trace(arg1); }'
```

Trace kernel module functions with flow indent (for example, `zfs` on Oracle Solaris):

```
dtrace -Fn 'fbt:zfs::'
```

Show kernel stack trace on function call (for example, `arc_read()`):

```
dtrace -n 'fbt::arc_read:entry { stack(); }'
```

Count kernel stack traces for a function call (for example, `arc_read()`):

```
dtrace -n 'fbt::arc_read:entry { @[stack()] = count(); }'
```

profile Provider

The profile provider can sample and count stack traces, which typically includes many stack frames for C code (frames from C++ and assembly may also be present in the stack trace).

Sample user stack trace at 101 Hertz, for a given PID:

```
dtrace -n 'profile-101 /pid == $target/ { @[ustack()] = count(); }' -p PID
```

Sample user stack trace at 101 Hertz, for processes named `example`:

```
dtrace -n 'profile-101 /execname == "example"/ { @[ustack()] = count(); }'
```

Sample user function at 101 Hertz, for a given PID:

```
dtrace -n 'profile-101 /pid == $target && arg1/ { @[ufunc(arg1)] = count(); }' -p PID
```

Sample kernel stack trace at 1001 Hertz:

```
dtrace -n 'profile-1001 { @[stack()] = count(); }'
```

Sample kernel stack trace at 1001 Hertz, for 10 seconds:

```
dtrace -n 'profile-1001 { @[stack()] = count(); } tick-10sec { exit(0); }'
```

Sample kernel function at 1001 Hertz:

```
dtrace -n 'profile-1001 /arg0/ { @[func(arg0)] = count(); }'
```

C One-Liners Selected Examples

Here we show examples of using several of the one-liners.

Trace Function Entry Arguments

Here the argument to libc's `strlen()` is traced. Since it is a pointer to a user-land address, it must be copied to the kernel address space for DTrace to print it, which we do using `copyinstr()`:

```
# dtrace -n 'pid$target:libc:strlen:entry { trace(copyinstr(arg0)); }' -c date
dtrace: description 'pid$target:libc:strlen:entry ' matched 1 probe
Mon Jul 12 03:41:40 UTC 2010
dtrace: pid 22835 has exited
CPU     ID                    FUNCTION:NAME
  0  96091                     strlen:entry   SUNW_OST_OSCMD
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   /usr/share/lib/zoneinfo
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   /usr/share/lib/zoneinfo
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   /usr/share/lib/zoneinfo
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   /usr/share/lib/zoneinfo
  0  96091                     strlen:entry   UTC
  0  96091                     strlen:entry   Mon Jul 12 03:41:40 UTC 2010
```

Each string that the `date(1)` command checked is visible, showing the full string in the output.

Show User Stack Trace on Function Call

Continuing with the previous example, the reason for `date(1)` calling `strlen()` can be determined by examining the user stack trace, fetched using `ustack()`:

```
# dtrace -n 'pid$target:libc:strlen:entry { ustack(); }' -c date
dtrace: description 'pid$target:libc:strlen:entry ' matched 1 probe
Mon Jul 12 03:55:24 UTC 2010
dtrace: pid 122838 has exited
CPU     ID                    FUNCTION:NAME
[...output truncated...]
  3  96091                     strlen:entry
```

continues

```
              libc.so.1`strlen
              libc.so.1`_ndoprnt+0x2370
              libc.so.1`snprintf+0x66
              libc.so.1`load_zoneinfo+0xc8
              libc.so.1`ltzset_u+0x177
              libc.so.1`mktime+0x1d9
              libc.so.1`__strftime_std+0x66
              libc.so.1`strftime+0x33
              date`main+0x1e5
              date`_start+0x7d

  3  96091                     strlen:entry
              libc.so.1`strlen
              libc.so.1`puts+0xdd
              date`main+0x1f2
              date`_start+0x7d
```

The output has been truncated so that only the last two stacks are shown. This shows that the last `strlen()` of `UTC` happened during `strftime()`, which was checking the time zone.

Count Kernel Function Calls Beginning With...

The ZFS file system is implemented as a C kernel module. This one-liner counts ZFS function calls beginning with `zfs_`:

```
# dtrace -n 'fbt::zfs_*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::zfs_*:entry ' matched 427 probes
^C

  zfs_copy_fuid_2_ace                                              2
  zfs_pathconf                                                     2
  zfs_groupmember                                                  4
  zfs_ioctl                                                        4
  zfs_ioc_objset_stats                                             5
  zfs_range_unlock_reader                                          6
  zfs_read                                                         6
  zfs_seek                                                         6
  zfs_getpage                                                     10
[...]
  zfs_ace_fuid_size                                             1894
  zfs_acl_next_ace                                              2442
  zfs_fastaccesschk_execute                                     3578
  zfs_lookup                                                    3656
```

The most frequently called function was `zfs_lookup()`, called 3,656 times while tracing.

See Also

For more one-liner examples, see Chapters 9 and 12 and other chapters for any one-liners that use the pid, fbt, and profile providers.

C Scripts

Many scripts in this book can examine C code execution; look in particular for those that use the fbt (kernel) and pid (user-land) providers. These include the scripts listed in Table 8-4.

C++

C++ is an object-oriented, general-purpose programming language developed initially to enhance the C language with new features such as objects and classes. C++ code is compiled into native, binary code for execution; like C, C++ code does not execute under an interpreter. C++ is a popular language and is used in many industries for software creation, both commercial software and customer-specific

Table 8-4 C Script Summary

Script	Description	Provider	Chapter
`scsirw.d`	Shows SCSI read/write stats, traced in the kernel	fbt	4
`zfssnoop.d`	Traces ZFS operations via kernel zfs module	fbt	5
`nfsv3fbtrws.d`	Traces NFSv3 operations via kernel nfs module	fbt	7
`getaddrinfo.d`	Shows latency of client `getaddrinfo()` lookups	pid	7
`uoncpu.d`	Profiles application on-CPU user stacks	profile	9
`uoffcpu.d`	Counts application off-CPU user stacks by time	sched	9
`plockstat`	User-level mutex and read/write lock statistics	plockstat	9
`mysqld_pid_qtime.d`	Traces mysqld and show query time distribution	pid	10
`libmysql_snoop.d`	Snoops client queries by tracing libmysqlclient	pid	10
`cuckoo.d`	Captures serial line sessions by tracing `cnwrite()`	fbt	11
`koncpu.d`	Profiles kernel on-CPU stacks	profile	12
`koffcpu.d`	Counts kernel off-CPU stacks by time	sched	12
`putnexts.d`	Streams `putnext()` tracing with stack back traces	fbt	12

custom applications. As native code, C++ lends itself well to instrumentation with DTrace; however, some of the features of C++, such as function name overloading, result in mangled function names when observing C++ code flow. Utilities, such as c++filt (part of the SunStudio compilers), can be used to improve the readability of the function names.

The tracing of C++ is the same as with C, with a couple of differences: Probe function names are C++ method signatures, and C++ objects cannot be walked as easily as C structures.

Function Names

C++ method names are represented as C++ signature strings in the probe function field. You can use wildcards to match only on the method name without specifying the entire signature string. For example, this matches the `DoURILoad()` C++ method from Mozilla Firefox 3.0:

```
solaris# dtrace -ln 'pid$target::*DoURILoad*:entry' -p `pgrep firefox-bin`
   ID   PROVIDER            MODULE                          FUNCTION NAME
118123  pid343704        libxul.so __1cKnsDocShellJDoURILoad6MpnGnsIURI_2ipnLnsISuppo
rts_pkcpnOnsIInputStream_8ippnLnsIDocShell_ppnKnsIRequest_ii_I_ entry
```

These signature strings can be passed to `c++filt` (or `gc++filt`) for readability:

```
solaris# dtrace -ln 'pid$target::*DoURILoad*:entry' -p `pgrep firefox-bin` | c++filt
   ID   PROVIDER            MODULE                          FUNCTION NAME
118123  pid343704        libxul.so unsigned nsDocShell::DoURILoad(nsIURI*,nsIURI*,int
,nsISupports*,const char*,nsIInputStream*,nsIInputStream*,int,nsIDocShell**,nsIRequest
**,int,int) entry
```

DTrace on Mac OS X post-processes the C++ signatures automatically.

Object Arguments

The arguments to C++ methods can be quite difficult to access from DTrace. The function entry probes provide the `arg0..N` variables, but the way these map to the the C++ arguments is up to the C++ compiler. `arg0` may be used for `this` (object pointer), and `arg1` onward are the method arguments (making them appear shifted compared to C). The compiler may also insert extra arguments for its own reasons.

Accessing data from within objects can be even trickier. If the offset is known (find out using a C++ debugger), it can be used to find the members. For user-land

C++, this will involve calling `copyin()` on `argN` variables + custom offsets. Or, try to find a function entry probe where the member of interest is available as an `argN` variable directly.

This is a case where DTrace makes something possible, but it isn't necessarily easy!

Java

The Java programming language is an object-oriented language that gets compiled into byte codes that are interpreted and executed by a Java virtual machine (JVM). The Java software development environment is extremely rich, with a large number of class libraries and extensions available, along with support on every conceivable platform—from cell phones and handheld devices to desktops and server systems running one of any mainstream available operating systems.

Starting with Java SE 6, the HotSpot VM makes available the hotspot and hotspot_jni providers to monitor JVM internal state and activities as well as the Java application that is running. All of the probes are USDT probes and are accessed using the process ID of the JVM process. The hotspot provider exposes the following types of probes:

- VM life-cycle probes
- Thread life-cycle probes
- Class-loading probes
- Garbage collection probes
- Method compilation probes
- Monitor probes
- Application-tracking probes

The Java SE 6 documentation[4] provides a full reference of all the probes and their arguments. It should be noted that string arguments are not guaranteed to be `NULL`-terminated. When string values are provided, they are always present as a pair: a pointer to the unterminated string and its length. Because of this, it is necessary to use `copyin()` with the correct length instead of `copyinstr()` for Java strings.

4. *http://download.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html*

If you want to observe Java-code-to-native-code interactions, you can use the hotspot_jni provider. This provider exposes probes for the entry/return points of all JNI functions. The name of the probe is the name of the JNI method, appended with `-entry` for entry probes and `-return` for return probes. The probe arguments correspond to the arguments provided to the JNI function[5] (in the case of the `*-entry` probes) or the return value (in the case of the `*-return` probes).

For the most part, the JVM probes exposed by hotspot and hotspot_jni providers are very lightweight and can be used on production machines. However, certain hotspot probes are expensive and turned off by default. These are the Java `method-entry`/`method-return`, `object-alloc`, and Java monitor probes. These probes require changes in the hotspot byte code interpreter and hotspot compiler (byte-code-to-machine-code compiler) and are comparatively costly even when disabled. To expose them all, the hotspot provider requires that the JVM be started with the `java -XX:+ExtendedDTraceProbes` command-line option:

```
java -XX:+ExtendedDTraceProbes -jar <jar file>
```

This facility can be turned on and off dynamically at runtime as well, using the `jinfo` utility. Here's an example:

```
jinfo -flag +ExtendedDTraceProbes <target JVM PID>
```

The `method-entry`/`method-return`, `object-alloc`, and `monitor` probes can also be selectively enabled via the Java command-line with the `-XX:+DTraceMethodProbes`, `-XX:+DTraceAllocProbes`, and `-XX:+DTraceMonitorProbes` options, respectively.

To see whether the hotspot provider is available, attempt to list probes using DTrace:

```
# dtrace -ln 'hotspot*:::'
   ID   PROVIDER          MODULE                       FUNCTION NAME
93669 hotspot_jni2642    libjvm.so   jni_GetStaticBooleanField GetStaticBooleanField-return
93670 hotspot_jni2642    libjvm.so      jni_GetStaticByteField GetStaticByteField-entry
93671 hotspot_jni2642    libjvm.so      jni_GetStaticByteField GetStaticByteField-return
93672 hotspot_jni2642    libjvm.so      jni_GetStaticCharField GetStaticCharField-entry
93673 hotspot_jni2642    libjvm.so      jni_GetStaticCharField GetStaticCharField-return
93674 hotspot_jni2642    libjvm.so    jni_GetStaticDoubleField GetStaticDoubleField-entry
93675 hotspot_jni2642    libjvm.so    jni_GetStaticDoubleField GetStaticDoubleField-return
```

5. The Invoke* methods are an exception. They omit the arguments that are passed to the Java method.

```
93676 hotspot_jni2642    libjvm.so        jni_GetStaticFieldID GetStaticFieldID-entry
93677 hotspot_jni2642    libjvm.so        jni_GetStaticFieldID GetStaticFieldID-return
[...]
```

More than 500 probes were listed, showing that the Java providers are available for PID 2642.

Example Java Code

The one-liners and scripts that follow are executed on the following example Java program.

Func_abc.java

This program demonstrates method flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each method also sleeps for one second, providing known method latency that can be examined.

```
public class Func_abc {
         public static void func_c() {
             System.out.println("Function C");
             try {
                 Thread.currentThread().sleep(1000);
             } catch (Exception e) { }
         }
         public static void func_b() {
             System.out.println("Function B");
             try {
                 Thread.currentThread().sleep(1000);
             } catch (Exception e) { }
             func_c();
         }
         public static void func_a() {
             System.out.println("Function A");
             try {
                 Thread.currentThread().sleep(1000);
             } catch (Exception e) { }
             func_b();
         }

         public static void main(String[] args) {
             func_a();
         }
     }
```

Java One-Liners

The Java one-liners here are organized by provider.

hotspot Provider

Count Java events:

```
dtrace -Zn 'hotspot*::: { @[probename] = count(); }'
```

Show Java activity by PID and UID:

```
dtrace -Zn 'hotspot*:::Call*-entry { @[pid, uid] = count(); }'
```

Much more is possible with the hotspot provider in D scripts, where strings can be retrieved, NULL terminated, and printed properly.

profile Provider

Profile Java stacks at 101 Hertz:

```
dtrace -Zn 'profile-101 /execname == "java"/ { @[jstack()] = count(); }'
```

Profile bigger Java stacks at 101 Hertz:

```
dtrace -x jstackstrsize=2048 -Zn 'profile-101 /execname == "java"/ { @[jstack()] =
count(); }'
```

Java One-Liners Selected Examples

This section provides selected one-liner Java examples.

Count Java events

Here the Func_abc example program was executed, without the Extended-DTraceProbes option to begin with:

```
# dtrace -Zn 'hotspot*::: { @[probename] = count(); }'
dtrace: description 'hotspot*::: ' matched 507 probes
^C

  AttachCurrentThread-entry                                    1
  AttachCurrentThread-return                                   1
  CallIntMethod-entry                                          1
  CallIntMethod-return                                         1
[…output truncated...]
  GetStringLength-entry                                       65
```

```
  GetStringLength-return                                                    65
  GetObjectField-entry                                                      67
  GetObjectField-return                                                     67
  DeleteLocalRef-entry                                                     112
  DeleteLocalRef-return                                                    112
  class-loaded                                                             327
```

There is still substantial visibility into the execution of Java. For this example, there were 327 class-loaded events.

Now here it is with `+ExtendedDTraceProbes`:

```
# dtrace -Zn 'hotspot*::: { @[probename] = count(); }'
dtrace: description 'hotspot*::: ' matched 507 probes
^C

  AttachCurrentThread-entry                                                  1
  AttachCurrentThread-return                                                 1
  CallIntMethod-entry                                                        1
  CallIntMethod-return                                                       1
[...output truncated...]
  GetStringLength-entry                                                     65
  GetStringLength-return                                                    65
  GetObjectField-entry                                                      67
  GetObjectField-return                                                     67
  DeleteLocalRef-entry                                                     112
  DeleteLocalRef-return                                                    112
  class-loaded                                                             327
  object-alloc                                                            5499
  method-return                                                          13432
  method-entry                                                           13439
```

Now method execution and object allocation are visible, with more than 13,000 method calls for this example.

Show Java Activity by PID and UID

Although this system is supposed to be idle, this one-liner has still counted 12 Java calls from PID 102642, UID 101. That process ID can now be examined with more DTrace or with other tools (for example, `pargs(1)`).

```
# dtrace -Zn 'hotspot*:::Call*-entry { @[pid, uid] = count(); }'
dtrace: description 'hotspot*:::Call*-entry ' matched 90 probes
^C

   102642      101                12
```

The probes that this one-liner uses are active even if the `ExtendedDTraceProbes` option has not been enabled.

Table 8-5 Java Script Summary

Script	Description	Provider
`j_calls.d`	Counts various Java events: method calls, object allocation, and so on	hotspot
`j_flow.d`	Traces method flow with indented output and time stamps	hotspot
`j_calltime.d`	Shows inclusive and exclusive method call times	hotspot
`j_thread.d`	Traces Java thread execution	hotspot

Java Scripts

The scripts included in Table 8-5 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

j_calls.d

This script counts various Java events, including method calls, object allocation, thread starts, and method compilation. Method calls and object allocation will be visible only if the `ExtendedDTraceProbes` option is set on the JVM.

Script

After copying in various strings, they are manually `NULL` terminated to the length provided by the probes (for example, lines 101, 103, and so on).

```
   1  #!/usr/sbin/dtrace -Zs
[...]
  48  #pragma D option quiet
  49
  50  dtrace:::BEGIN
  51  {
  52        printf("Tracing... Hit Ctrl-C to end.\n");
  53  }
  54
  55  hotspot*:::method-entry
  56  {
  57        this->class = (char *)copyin(arg1, arg2 + 1);
  58        this->class[arg2] = '\0';
  59        this->method = (char *)copyin(arg3, arg4 + 1);
  60        this->method[arg4] = '\0';
  61        this->name = strjoin(strjoin(stringof(this->class), "."),
  62            stringof(this->method));
  63        @calls[pid, "method", this->name] = count();
  64  }
  65
  66  hotspot*:::object-alloc
  67  {
  68        this->class = (char *)copyin(arg1, arg2 + 1);
  69        this->class[arg2] = '\0';
```

```
      @calls[pid, "oalloc", stringof(this->class)] = count();
}

hotspot*:::class-loaded
{
      this->class = (char *)copyin(arg0, arg1 + 1);
      this->class[arg1] = '\0';
      @calls[pid, "cload", stringof(this->class)] = count();
}

hotspot*:::thread-start
{
      this->thread = (char *)copyin(arg0, arg1 + 1);
      this->thread[arg1] = '\0';
      @calls[pid, "thread", stringof(this->thread)] = count();
}

hotspot*:::method-compile-begin
{
      this->class = (char *)copyin(arg0, arg1 + 1);
      this->class[arg1] = '\0';
      this->method = (char *)copyin(arg2, arg3 + 1);
      this->method[arg3] = '\0';
      this->name = strjoin(strjoin(stringof(this->class), "."),
          stringof(this->method));
      @calls[pid, "mcompile", this->name] = count();
}

hotspot*:::compiled-method-load
{
      this->class = (char *)copyin(arg0, arg1 + 1);
      this->class[arg1] = '\0';
      this->method = (char *)copyin(arg2, arg3 + 1);
      this->method[arg3] = '\0';
      this->name = strjoin(strjoin(stringof(this->class), "."),
          stringof(this->method));
      @calls[pid, "mload", this->name] = count();
}

dtrace:::END
{
      printf(" %6s %-8s %-52s %8s\n", "PID", "TYPE", "NAME", "COUNT");
      printa(" %6d %-8s %-52s %@8d\n", @calls);
}
```

The `TYPE` column prints the event type:

`cload`: Class load

`method`: Method call

`mcompile`: Method compile

`mload`: Compiled method load

`oalloc`: Object allocation

`thread`: Thread start

Example

The `j_calls.d` script was used to trace Java events from the example `Func_abc` program:

```
# j_calls.d
Tracing... Hit Ctrl-C to end.
^C

    PID TYPE      NAME                                                      COUNT
 311334 cload     Func_abc                                                      1
 311334 cload     java/io/BufferedInputStream                                   1
 311334 cload     java/io/BufferedOutputStream                                  1
 311334 cload     java/io/BufferedReader                                        1
 311334 cload     java/io/BufferedWriter                                        1
 311334 cload     java/io/Closeable                                             1
 311334 cload     java/io/Console                                               1
[...output truncated...]
 311334 method    java/lang/String.substring                                   94
 311334 method    java/util/Arrays.copyOfRange                                107
 311334 method    java/lang/String.getChars                                   156
 311334 method    java/lang/System.getSecurityManager                         174
 311334 method    java/lang/String.<init>                                     175
 311334 method    java/lang/String.equals                                     202
 311334 method    java/lang/Math.min                                          208
 311334 method    java/lang/String.hashCode                                   213
 311334 method    java/lang/String.indexOf                                    302
 311334 oalloc    [Ljava/lang/Object;                                         326
 311334 method    java/lang/System.arraycopy                                  360
 311334 oalloc    [I                                                          374
 311334 oalloc    java/lang/Class                                             395
 311334 oalloc    [B                                                          406
 311334 oalloc    [S                                                          486
 311334 method    java/lang/StringBuilder.append                              533
 311334 oalloc    [[I                                                         541
 311334 method    java/lang/AbstractStringBuilder.append                      549
 311334 method    java/lang/Object.<init>                                     823
 311334 oalloc    java/lang/String                                            931
 311334 oalloc    [C                                                         1076
 311334 method    java/lang/String.charAt                                    1960
```

More than 1,000 lines of output were truncated. The most frequent event was 1,960 method calls for `java/lang/String.charAt` and 1,076 object allocations of type `[C`.

j_flow.d

This script traces Java execution showing method flow as an indented output, with time stamps. Since this traces method calls, the `ExtendedDTraceProbes` option must be set on the JVM for this script to work.

Script

A stack depth variable, `self->depth`, is maintained so that indentation can be printed. Apart from being a thread-local variable, it is also keyed on the Java thread identifier (arg0) to maintain a separate indentation between different running Java threads. If desired, the script could be enhanced to include this thread identifier as a column in the output.

```
#!/usr/sbin/dtrace -Zs
[...]
/* increasing bufsize can reduce drops */
#pragma D option bufsize=16m
#pragma D option quiet
#pragma D option switchrate=10

self int depth[int];

dtrace:::BEGIN
{
        printf("%3s %6s %-16s -- %s\n", "C", "PID", "TIME(us)", "CLASS.METHOD");
}

hotspot*:::method-entry
{
        this->class = (char *)copyin(arg1, arg2 + 1);
        this->class[arg2] = '\0';
        this->method = (char *)copyin(arg3, arg4 + 1);
        this->method[arg4] = '\0';

        printf("%3d %6d %-16d %*s-> %s.%s\n", cpu, pid, timestamp / 1000,
            self->depth[arg0] * 2, "", stringof(this->class),
            stringof(this->method));
        self->depth[arg0]++;
}

hotspot*:::method-return
{
        this->class = (char *)copyin(arg1, arg2 + 1);
        this->class[arg2] = '\0';
        this->method = (char *)copyin(arg3, arg4 + 1);
        this->method[arg4] = '\0';

        self->depth[arg0] -= self->depth[arg0] > 0 ? 1 : 0;
        printf("%3d %6d %-16d %*s<- %s.%s\n", cpu, pid, timestamp / 1000,
            self->depth[arg0] * 2, "", stringof(this->class),
            stringof(this->method));
}
```

Example

This traces the example Func_abc program:

```
# j_flow.d
  C    PID TIME(us)         -- CLASS.METHOD
  0 311403 4789112583163    -> java/lang/Object.<clinit>
  0 311403 4789112583207      -> java/lang/Object.registerNatives
  0 311403 4789112583323      <- java/lang/Object.registerNatives
  0 311403 4789112583333    <- java/lang/Object.<clinit>
  0 311403 4789112583343    -> java/lang/String.<clinit>
  0 311403 4789112583732      -> java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583743        -> java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583752          -> java/lang/Object.<init>
```

continues

```
  0 311403 4789112583760          <- java/lang/Object.<init>
  0 311403 4789112583767        <- java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583774      <- java/lang/String$CaseInsensitiveComparator.<init>
  0 311403 4789112583783    <- java/lang/String.<clinit>
  0 311403 4789112583849    -> java/lang/System.<clinit>
  0 311403 4789112583859      -> java/lang/System.registerNatives
  0 311403 4789112583878      <- java/lang/System.registerNatives
  0 311403 4789112583887      -> java/lang/System.nullInputStream
  0 311403 4789112583895        -> java/lang/System.currentTimeMillis
  0 311403 4789112583905        <- java/lang/System.currentTimeMillis
[...]
```

The output was more than 1,000 lines long. To see the functions from the `Func_abc` program, the output was saved to a file that was filtered using `grep(1)`:

```
# grep Func_abc outputfile
  0 311403 4789112982182    -> Func_abc.main
  0 311403 4789112982193      -> Func_abc.func_a
  0 311403 4789113990080        -> Func_abc.func_b
  0 311403 4789115000081          -> Func_abc.func_c
  0 311403 4789116010073          <- Func_abc.func_c
  0 311403 4789116010080        <- Func_abc.func_b
  0 311403 4789116010086      <- Func_abc.func_a
  0 311403 4789116010093    <- Func_abc.main
```

This shows the expected function flow and time stamp jumps. (Note that small time stamp jumps of less than 10 us may be dominated by the DTrace probe effect of both tracing the method probes and calling `copyin()` for the class and method names.)

The time stamps can also be used for postsorting the output, which may become shuffled on multi-CPU systems.

See Also: j_classflow.d

The DTraceToolkit contains a variant of `j_flow.d` called `j_classflow.d` (not included here), which only traces the given class. Here's an example:

```
# j_classflow.d Func_abc
  C    PID TIME(us)         -- CLASS.METHOD
  0 311425 4789778117827    -> Func_abc.main
  0 311425 4789778117844      -> Func_abc.func_a
  0 311425 4789779120071        -> Func_abc.func_b
  0 311425 4789780130070          -> Func_abc.func_c
  0 311425 4789781140067          <- Func_abc.func_c
  0 311425 4789781140079        <- Func_abc.func_b
  0 311425 4789781140087      <- Func_abc.func_a
  0 311425 4789781140095    <- Func_abc.main
^C
```

This avoids the need to output to a file for later filtering, if desired.

j_calltime.d

This script traces the time taken by Java methods and garbage collection, and prints a report. The times for functions are as follows:

Inclusive: Showing the elapsed time for methods

Exclusive: Showing which excludes time spent in other called methods

This can be used for performance analysis of Java programs to identify what is responsible for latency.

Script

To associate method entries to returns so that delta times can be calculated, the `self->exclude` and `self->method` variables are keyed on both the Java thread ID and our own maintained stack depth, `self->depth`.

```
#!/usr/sbin/dtrace -Zs
[...]
#define TOP 10          /* default output truncation */
#define B_FALSE   0

#pragma D option quiet
#pragma D option defaultargs

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
      top = $1 != 0 ? $1 : TOP;
}

hotspot*:::method-entry
{
      self->depth[arg0]++;
      self->exclude[arg0, self->depth[arg0]] = 0;
      self->method[arg0, self->depth[arg0]] = timestamp;
}

hotspot*:::method-return
/self->method[arg0, self->depth[arg0]]/
{
      this->elapsed_incl = timestamp - self->method[arg0, self->depth[arg0]];
      this->elapsed_excl = this->elapsed_incl -
          self->exclude[arg0, self->depth[arg0]];
      self->method[arg0, self->depth[arg0]] = 0;
      self->exclude[arg0, self->depth[arg0]] = 0;

      this->class = (char *)copyin(arg1, arg2 + 1);
      this->class[arg2] = '\0';
      this->method = (char *)copyin(arg3, arg4 + 1);
      this->method[arg4] = '\0';
      this->name = strjoin(strjoin(stringof(this->class), "."),
          stringof(this->method));

      @num[pid, "method", this->name] = count();
```

continues

```
        @num[0, "total", "-"] = count();
        @types_incl[pid, "method", this->name] = sum(this->elapsed_incl);
        @types_excl[pid, "method", this->name] = sum(this->elapsed_excl);
        @types_excl[0, "total", "-"] = sum(this->elapsed_excl);

        self->depth[arg0]--;
        self->exclude[arg0, self->depth[arg0]] += this->elapsed_incl;
}

hotspot*:::gc-begin
{
        self->gc = timestamp;
        self->full = (boolean_t)arg0;
}

hotspot*:::gc-end
/self->gc/
{
        this->elapsed = timestamp - self->gc;
        self->gc = 0;

        @num[pid, "gc", self->full == B_FALSE ? "GC" : "Full GC"] = count();
        @types[pid, "gc", self->full == B_FALSE ? "GC" : "Full GC"] =
            sum(this->elapsed);
        self->full = 0;
}

dtrace:::END
{
        trunc(@num, top);
        printf("\nTop %d counts,\n", top);
        printf("   %6s %-10s %-48s %8s\n", "PID", "TYPE", "NAME", "COUNT");
        printa("   %6d %-10s %-48s %@8d\n", @num);

        trunc(@types, top);
        normalize(@types, 1000);
        printf("\nTop %d elapsed times (us),\n", top);
        printf("   %6s %-10s %-48s %8s\n", "PID", "TYPE", "NAME", "TOTAL");
        printa("   %6d %-10s %-48s %@8d\n", @types);

        trunc(@types_excl, top);
        normalize(@types_excl, 1000);
        printf("\nTop %d exclusive method elapsed times (us),\n", top);
        printf("   %6s %-10s %-48s %8s\n", "PID", "TYPE", "NAME", "TOTAL");
        printa("   %6d %-10s %-48s %@8d\n", @types_excl);

        trunc(@types_incl, top);
        normalize(@types_incl, 1000);
        printf("\nTop %d inclusive method elapsed times (us),\n", top);
        printf("   %6s %-10s %-48s %8s\n", "PID", "TYPE", "NAME", "TOTAL");
        printa("   %6d %-10s %-48s %@8d\n", @types_incl);
}
```

Example

The example program `Func_abc` was traced:

```
# j_calltime.d
Tracing... Hit Ctrl-C to end.
^C
```

```
Top 10 counts,
      PID TYPE       NAME                                              COUNT
   347032 method     java/lang/String.equals                             221
   347032 method     java/lang/String.hashCode                           230
   347032 method     java/lang/Math.min                                  233
   347032 method     java/lang/String.indexOf                            314
   347032 method     java/lang/System.arraycopy                          397
   347032 method     java/lang/StringBuilder.append                      658
   347032 method     java/lang/AbstractStringBuilder.append              676
   347032 method     java/lang/Object.<init>                             874
   347032 method     java/lang/String.charAt                            2285
        0 total      -                                                 13428

Top 10 elapsed times (us),
      PID TYPE       NAME                                              TOTAL

Top 10 exclusive method elapsed times (us),
      PID TYPE       NAME                                              TOTAL
   347032 method     java/lang/System.initProperties                    3490
   347032 method     java/util/Arrays.copyOf                            3777
   347032 method     java/lang/String.charAt                            3919
   347032 method     java/lang/AbstractStringBuilder.append             4784
   347032 method     java/lang/String.<init>                            5860
   347032 method     java/lang/StringBuilder.append                    11556
   347032 method     sun/net/www/ParseUtil.decode                      14009
   347032 method     java/io/UnixFileSystem.normalize                  14635
   347032 method     java/lang/Thread.sleep                          3019529
        0 total      -                                               3307655

Top 10 inclusive method elapsed times (us),
      PID TYPE       NAME                                              TOTAL
   347032 method     sun/misc/Launcher$AppClassLoader.loadClass       103271
   347032 method     java/lang/ClassLoader.loadClassInternal          103597
   347032 method     sun/misc/URLClassPath.getLoader                  122309
   347032 method     java/security/AccessController.doPrivileged      267620
   347032 method     java/lang/ClassLoader.loadClass                  276966
   347032 method     Func_abc.func_c                                 1010318
   347032 method     Func_abc.func_b                                 2020055
   347032 method     java/lang/Thread.sleep                          3019529
   347032 method     Func_abc.func_a                                 3029564
   347032 method     Func_abc.main                                   3029591
```

The difference between inclusive and exclusive method times is demonstrated by the example program: `func_a()` had 3.03 seconds of inclusive time in total but did not make the top ten exclusive times when its subcalls (`Thread.sleep()`) were excluded. The top exclusive time was `Thread.sleep()`, where the actual time was spent waiting.

For this example, there was nothing in the "Top 10 elapsed times (us)" summary, which only includes garbage collect events.

Note

`j_calltime.d` traces all `method-entry` and `method-return` probes, which can be CPU expensive when fired frequently, slowing down the target application (for example, by 10x!). This is most evident for Java programs that call many thousands of methods per second, where the results from `j_calltime.d` can be noticeably inflated by the probe overhead from DTrace. Also, methods that call many thousands of submethods will be slowed down further than methods that do not, which means the results are not only inflated but also skewed. Use the profile provider and `jstack()` to double-check any findings (see the one-liners).

See Also: j_calldist.d

There is a variant of `j_calltime.d` in the DTraceToolkit called `j_calldist.d` (not included here), which prints times as distribution plots by subroutine name. Its functionality is similar to the Perl version, `pl_calltime.d`, which is demonstrated in the "Perl Scripts" section under `pl_callinfo.d`.

See Also: j_cputime.d, j_cpudist.d

Also in the DTraceToolkit are variants of the previous two scripts that trace on-CPU time instead of elapsed time. This time serves a different role: Elapsed time latency can include I/O wait time for system resources (disk, network), whereas latency that is on-CPU time is reflective of the time to process the Java code. Their functionality is similar to the Perl versions: `pl_cputime.d` is demonstrated in the "Perl Scripts" section under `pl_calltime.d`.

Since the vtimestamps that these tools use attempt to negate the DTrace probe effect, the times reported may be more accurate than `j_calltime.d`, especially when methods are called frequently.

j_thread.d

This script traces Java thread execution, showing time, PID, TID, and thread name.

Script

Here's the script, with the heading comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

dtrace:::BEGIN
{
      printf("%-20s  %6s/%-5s -- %s\n", "TIME", "PID", "TID", "THREAD");
}

hotspot*:::thread-start
{
      this->thread = (char *)copyin(arg0, arg1 + 1);
      this->thread[arg1] = '\0';
      printf("%-20Y  %6d/%-5d => %s\n", walltimestamp, pid, tid,
          stringof(this->thread));
}

```

```
hotspot*:::thread-stop
{
        this->thread = (char *)copyin(arg0, arg1 + 1);
        this->thread[arg1] = '\0';
        printf("%-20Y  %6d/%-5d <= %s\n", walltimestamp, pid, tid,
            stringof(this->thread));
}
```

Example

Java thread execution from the example `Func_abc` program was examined using `j_thread.d`:

```
# j_thread.d
TIME                     PID/TID   -- THREAD
2010 Jul 11 04:21:33  346986/4     => Reference Handler
2010 Jul 11 04:21:33  346986/5     => Finalizer
2010 Jul 11 04:21:33  346986/6     => Signal Dispatcher
2010 Jul 11 04:21:33  346986/7     => CompilerThread0
2010 Jul 11 04:21:33  346986/8     => CompilerThread1
2010 Jul 11 04:21:33  346986/9     => Low Memory Detector
2010 Jul 11 04:21:36  346986/6     <= Signal Dispatcher
^C
```

The threads started (=>) can be seen in the output for PID 346986, which was the `Func_abc` program. Only one thread exit is seen (<=), because the JVM exited when the program stopped so that the thread stop probes were not fired.

See Also

There are other scripts in the DTraceToolkit for tracing Java in the `/Java` directory:

- `j_stat.d`: A stat-style tool for Java events
- `j_package.d`: Count Java class loads by package
- `j_objnew.d`: Object allocation report

JavaScript

JavaScript is an object-oriented scripting language that initially evolved to facilitate embedding executable code in Web pages to add new features and functionality to Web sites. Because of this history, much of the JavaScript code you will find comes in the form of client-side programs that execute within a Web browser. However, JavaScript applications are increasingly popular outside the context of Web browsing.

The Spider Monkey JavaScript engine for Mozilla Firefox has been instrumented with a javascript DTrace provider[6] as part of a suite of DTrace providers for Mozilla.[7] It was integrated into the Mozilla source for "Bug 388564 – (jsdtrace) [RFE] JavaScript Tracing Framework"[8] in October 2007. Firefox must be compiled with the `--enable-dtrace` option for the provider to be present; you'll need to be familiar with source code compilation to do this yourself. Firefox packages with the javascript provider enabled are available for Oracle Solaris, from the Sun contributed builds site.[9]

To check that the javascript provider is available, attempt to list probes while a browser is running:

```
# dtrace -ln 'javascript*:::'
   ID   PROVIDER        MODULE                         FUNCTION NAME
10982 javascript26526   libmozjs.so                  js_Interpret function-info
10983 javascript26526   libmozjs.so      jsdtrace_function_return function-return
10984 javascript26526   libmozjs.so                  js_Interpret function-return
10985 javascript26526   libmozjs.so        jsdtrace_function_rval function-rval
10986 javascript26526   libmozjs.so                  js_Interpret function-rval
10987 javascript26526   libmozjs.so        jsdtrace_object_create object-create
10988 javascript26526   libmozjs.so    js_NewObjectWithGivenProto object-create
10989 javascript26526   libmozjs.so   jsdtrace_object_create_done object-create-done
55235 javascript26526   libmozjs.so         jsdtrace_execute_done execute-done
55236 javascript26526   libmozjs.so                    js_Execute execute-done
88079 javascript26526   libmozjs.so    js_NewObjectWithGivenProto object-create-done
88080 javascript26526   libmozjs.so  jsdtrace_object_create_start object-create-start
88081 javascript26526   libmozjs.so    js_NewObjectWithGivenProto object-create-start
88082 javascript26526   libmozjs.so      jsdtrace_object_finalize object-finalize
88083 javascript26526   libmozjs.so             js_FinalizeObject object-finalize
93947 javascript26526   libmozjs.so        jsdtrace_execute_start execute-start
93948 javascript26526   libmozjs.so                    js_Execute execute-start
93949 javascript26526   libmozjs.so        jsdtrace_function_args function-args
93950 javascript26526   libmozjs.so                  js_Interpret function-args
93951 javascript26526   libmozjs.so       jsdtrace_function_entry function-entry
93952 javascript26526   libmozjs.so                  js_Interpret function-entry
93953 javascript26526   libmozjs.so        jsdtrace_function_info function-info
```

This output shows that there is a javascript provider for process ID 26526 and shows the probe names (`NAME` column) along with their location in the libmozjs source (`FUNCTION` column).

6. This was originally written by Brendan Gregg and later developed as part of a Mozilla DTrace provider suite by engineers from both Sun and Mozilla.

7. This is currently at *www.opensolaris.org/os/project/mozilla-dtrace*. Also see "Bug 370906 – (dtrace) [RFE] Dynamic Tracing Framework for Mozilla" at *https://bugzilla.mozilla.org/show_bug.cgi?id=370906*.

8. *https://bugzilla.mozilla.org/show_bug.cgi?id=388564*

9. *http://releases.mozilla.com/sun/solaris10/*

The DTrace javascript provider interface is as follows:

```
provider javascript {
    probe function-entry(file, class, func)
    probe function-info(file, class, func, lineno, runfile, runlineno)
    probe function-args(file, class, func, argc, argv, argv0, argv1,
        argv2, argv3, argv4)
    probe function-rval(file, class, func, lineno, rval, rval0)
    probe function-return(file, class, func)
    probe object-create-start(file, class)
    probe object-create(file, class, *object, rlineno)
    probe object-create-done(file, class)
    probe object-finalize(NULL, class, *object)
    probe execute-start(file, lineno)
    probe execute-done(file, lineno)
};
```

If this interface has changed for the javascript provider version you are using, update the one-liners and scripts that follow accordingly.

This section demonstrates the javascript provider as shipped in Mozilla Firefox 3.0 and executed on Oracle Solaris by the Sun contributed build of Firefox 3.0, which enables the DTrace javascript provider by default.

Example JavaScript Code

The one-liners and scripts that follow are executed on the following example JavaScript program.

func_clock.html

This program demonstrates function flow and on-CPU time: `func_a()` calls `func_b()`, which calls `func_c()`. Each function executes code in a loop a different number of times so that the time spent on-CPU executing of code can be examined and compared.

```
<HTML>
<HEAD>
<TITLE>func_clock, JavaScript</TITLE>
<SCRIPT type="text/javascript">
function func_c() {
      document.getElementById('now').innerHTML += "Function C<br>"
      for (i = 0; i < 30000; i++) {
            j = i + 1
      }
 }

 function func_b() {
       document.getElementById('now').innerHTML += "Function B<br>"
       for (i = 0; i < 20000; i++) {
```

continues

```
                j = i + 1
            }
            func_c()
    }

    function func_a() {
            document.getElementById('now').innerHTML += "Function A<br>"
            for (i = 0; i < 10000; i++) {
                j = i + 1
            }
            func_b()
    }

    function start() {
            now = new Date()
            document.getElementById('now').innerHTML = now + "<br>"
            func_a()
            var timeout = setTimeout('start()', 1000)
    }
    </SCRIPT>
    </HEAD>
    <BODY onload="start()">
    <DIV id="now"></DIV>
    </BODY>
    </HTML>
```

This is similar to other example programs in this chapter; however, here `func_a()` is executed every second from the `start()` function timer, because the script updates an on-screen clock. This allows the JavaScript program to be left running in the browser and analyzed with DTrace, as demonstrated by examples in this section, without needing to reload the page. Reloading a page can fire numerous JavaScript routines built into Firefox, which would clutter every example (this is demonstrated in the one-liner examples for reference).

JavaScript One-Liners

This section provide JavaScript one-liners.

javascript Provider

Trace program execution showing filename and line number:

```
dtrace -n 'javascript*:::execute-start { printf("%s:%d", copyinstr(arg0), arg1); }'
```

Trace function calls showing function name:

```
dtrace -n 'javascript*:::function-entry { trace(copyinstr(arg2)); }'
```

Count function calls by function name:

```
dtrace -n 'javascript*:::function-entry { @[copyinstr(arg2)] = count(); }'
```

Count function calls by function filename:

```
dtrace -n 'javascript*:::function-entry { @[copyinstr(arg0)] = count(); }'
```

Count object creation by object class:

```
dtrace -n 'javascript*:::object-create { @[copyinstr(arg1)] = count(); }'
```

Object entropy stat:

```
dtrace -n 'javascript*:::object-create { @ = sum(1); } javascript*:::object-finalize
{ @ = sum(-1); } tick-10s { printa("%@d", @); }'
```

JavaScript One-Liners Selected Examples

This section provides selected JavaScript one-liner examples.

Trace Program Execution Showing Filename and Line Number

This one-liner was used to trace the execution of `func_clock.html`:

```
# dtrace -n 'javascript*:::execute-start { printf("%s:%d", copyinstr(arg0), arg1); }'
dtrace: description 'javascript*:::execute-start ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1 55232 jsdtrace_execute_start:execute-start file:///home/brendan/js/func_clock.html
:32
  1 55232 jsdtrace_execute_start:execute-start file:///home/brendan/js/func_clock.html
:32
  1 55232 jsdtrace_execute_start:execute-start file:///home/brendan/js/func_clock.html
:32
^C
```

A line of output was printed every second as the `func_clock.html` JavaScript executed line 32: the timeout function `start()`.

Trace Function Calls Showing Function Name

The execution of `func_clock.html` is traced using this one-liner:

```
# dtrace -n 'javascript*:::function-entry { trace(copyinstr(arg2)); }'
dtrace: description 'javascript*:::function-entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  55236 jsdtrace_function_entry:function-entry   start
  1  55236 jsdtrace_function_entry:function-entry   getElementById
  1  55236 jsdtrace_function_entry:function-entry   func_a
  1  55236 jsdtrace_function_entry:function-entry   getElementById
  1  55236 jsdtrace_function_entry:function-entry   func_b
  1  55236 jsdtrace_function_entry:function-entry   getElementById
  1  55236 jsdtrace_function_entry:function-entry   func_c
  1  55236 jsdtrace_function_entry:function-entry   getElementById
  1  55236 jsdtrace_function_entry:function-entry   setTimeout
[...]
```

The output repeats for every update of the clock, showing the functions that were executed.

Count Function Calls by Function Filename

The filename for function calls was counted by this one-liner, which was executed for three seconds:

```
# dtrace -n 'javascript*:::function-entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'javascript*:::function-entry ' matched 2 probes
^C

  file:///home/brendan/js/func_clock.html                          27
```

This has identified `func_clock.html` as the source of all the function calls, along with the full URL (it was accessed locally). The function names (nine per second) were traced earlier by the previous one-liner.

Now the same JavaScript program in `func_clock.html` was traced, but instead of analyzing it while it is already running, the page is reloaded by clicking the browser reload button:

```
# dtrace -n 'javascript*:::function-entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'javascript*:::function-entry ' matched 2 probes
^C

  chrome://global/content/bindings/text.xml                         1
  chrome://global/content/viewZoomOverlay.js                        1
  chrome://global/content/bindings/findbar.xml                      2
  chrome://global/content/bindings/textbox.xml                      2
  chrome://global/content/bindings/autocomplete.xml                 4
  chrome://global/content/bindings/toolbar.xml                      4
```

```
    chrome://global/content/bindings/general.xml                     5
    chrome://global/content/bindings/popup.xml                       5
    file:///opt/sfw/lib/firefox3/components/nsContentPrefService.js              5
    file:///opt/sfw/lib/firefox3/components/nsLoginManager.js              5
    file:///opt/sfw/lib/firefox3/modules/utils.js                    5
    file:///opt/sfw/lib/firefox3/modules/XPCOMUtils.jsm               6
    chrome://global/content/bindings/button.xml                      8
    file:///opt/sfw/lib/firefox3/components/nsSessionStore.js              8
    chrome://reporter/content/reporterOverlay.js                     9
    chrome://global/content/bindings/progressmeter.xml              12
    XStringBundle                                                   14
    chrome://global/content/bindings/browser.xml                    22
    file:///home/brendan/js/func_clock.html                         36
    chrome://browser/content/tabbrowser.xml                         42
    chrome://browser/content/browser.js                            120
```

The execution of JavaScript for `chrome` has been traced, which is the user interface for Mozilla Firefox[10] and was triggered by clicking the reload button. This is why a continually running clock has been used for the examples in this section: Its execution can be traced without reloading and hence without cluttering the example with Chrome events.

Object Entropy Stat

For this example, we traced a Web browser with numerous tabs open, running various JavaScript programs from the Web. The source of the JavaScript can be identified by the previous one-liners; here, we are interested in object creation, in particular, the leaking of objects. This one-liner increments an aggregation whenever an object is created and decrements it when an object is freed (finalized):

```
# dtrace -n 'javascript*:::object-create { @ = sum(1); } javascript*:::object-
finalize
{ @ = sum(-1); } tick-10s { printa("%@d", @); }'
dtrace: description 'javascript*:::object-create ' matched 5 probes
CPU     ID                    FUNCTION:NAME
  1  91387                        :tick-10s 2591
  1  91387                        :tick-10s 4738
  1  91387                        :tick-10s 7324
  1  91387                        :tick-10s 15
  1  91387                        :tick-10s 2603
  1  91387                        :tick-10s 4744
  1  91387                        :tick-10s 7334
  1  91387                        :tick-10s 14
  1  91387                        :tick-10s 2600
  1  91387                        :tick-10s 4752
  1  91387                        :tick-10s 7332
  1  91387                        :tick-10s -5
  1  91387                        :tick-10s 2498
[...]
```

10. *https://developer.mozilla.org/en/Chrome*

Table 8-6 JavaScript Script Summary

Script	Description	Provider
`js_calls.d`	Counts function calls by subroutine name	javascript
`js_flowinfo.d`	Traces function flow with indented output	javascript
`js_calltime.d`	Shows inclusive and exclusive function call times	javascript

The count has become negative because the one-liner was executed after some objects had already been created, which were then freed. The output shows that objects are being freed every 40 seconds and that there does not appear to be an upward trend (leak).

JavaScript Scripts

The scripts included in Table 8-6 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

js_calls.d

This script counts JavaScript events: function calls, program execution, and object creation and destruction.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing JavaScript... Hit Ctrl-C to end.\n");
}

javascript*:::function-entry
{
      this->name = copyinstr(arg2);
        @calls[basename(copyinstr(arg0)), "func", this->name] = count();
}

javascript*:::execute-start
{
      this->filename = basename(copyinstr(arg0));
        @calls[this->filename, "exec", "."] = count();
}

javascript*:::object-create-start
{
```

```
        this->name = copyinstr(arg1);
        this->filename = basename(copyinstr(arg0));
          @calls[this->filename, "obj-new", this->name] = count();
}

javascript*:::object-finalize
{
        this->name = copyinstr(arg1);
          @calls["<null>", "obj-free", this->name] = count();
}

dtrace:::END
{
          printf(" %-24s %-10s %-30s %8s\n", "FILE", "TYPE", "NAME", "CALLS");
          printa(" %-24s %-10s %-30s %@8d\n", @calls);
}
```

Example

Here the `js_calls.d` script traced the execution of JavaScript for five seconds by providing a `tick-5sec` action at the command line (`-n`):

```
# js_calls.d -n 'tick-5sec { exit(0); }'
Tracing JavaScript... Hit Ctrl-C to end.
 FILE                     TYPE       NAME                                CALLS
 func_clock.html          exec       .                                       5
 func_clock.html          func       func_a                                  5
 func_clock.html          func       func_b                                  5
 func_clock.html          func       func_c                                  5
 func_clock.html          func       setTimeout                              5
 func_clock.html          func       start                                   5
 func_clock.html          obj-new    Date                                    5
 func_clock.html          func       getElementById                         20
```

The output shows five updates to the clock: capturing the execution of JavaScript code from `func_clock.html` five times (`TYPE "exec"`) and the functions from that program. Five `Date` objects were also created while tracing, and none were freed.

js_flowinfo.d

This program traces JavaScript function flow, printing various details.

Script

This script uses the `function-info` probe so that the file and line number from which functions were executed can be printed (`arg4`, `arg5`). This is different from the source file and line number where the functions were declared (which `function-info` provides as `arg0` and `arg3`).

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

self int depth;

dtrace:::BEGIN
{
      printf("%3s %6s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "DELTA(us)",
          "FILE", "LINE", "TYPE", "FUNC");
}

javascript*:::function-info,
javascript*:::function-return
/self->last == 0/
{
      self->last = timestamp;
}

javascript*:::function-info
{
      this->delta = (timestamp - self->last) / 1000;
      printf("%3d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid,
          this->delta, basename(copyinstr(arg4)), arg5, "func",
          self->depth * 2, "", copyinstr(arg2));
      self->depth++;
      self->last = timestamp;
}

javascript*:::function-return
{
      this->delta = (timestamp - self->last) / 1000;
      self->depth -= self->depth > 0 ? 1 : 0;
      printf("%3d %6d %10d  %16s:-    %-8s %*s<- %s\n", cpu, pid,
          this->delta, basename(copyinstr(arg0)), "func", self->depth * 2,
          "", copyinstr(arg2));
      self->last = timestamp;
}
```

The `TYPE` column will only ever contain `function` for JavaScript functions. It's been included in case you want to enhance this script to include other event types, such as internal libmosjs execution, libxul execution (main Firefox library), system calls, and so on, which can then be examined in the context of the JavaScript program.

Example

The `func_clock.html` program was traced using `js_flowinfo.d`:

```
# js_flowinfo.d
  C    PID  DELTA(us)              FILE:LINE TYPE     -- FUNC
  1  43704          2   func_clock.html:32   func     -> start
  1  43704         40   func_clock.html:30   func       -> getElementById
  1  43704         56   func_clock.html:-    func       <- getElementById
  1  43704        486   func_clock.html:31   func       -> func_a
  1  43704         14   func_clock.html:21   func         -> getElementById
  1  43704         19   func_clock.html:-    func         <- getElementById
```

```
   1  43704       5602   func_clock.html:25   func          -> func_b
   1  43704         12   func_clock.html:13   func            -> getElementById
   1  43704         19   func_clock.html:-    func            <- getElementById
   1  43704      11197   func_clock.html:17   func            -> func_c
   1  43704         15   func_clock.html:6    func              -> getElementById
   1  43704         23   func_clock.html:-    func              <- getElementById
   1  43704      16714   func_clock.html:-    func            <- func_c
   1  43704         12   func_clock.html:-    func          <- func_b
   1  43704          9   func_clock.html:-    func        <- func_a
   1  43704         11   func_clock.html:32   func        -> setTimeout
   1  43704        181   func_clock.html:-    func        <- setTimeout
   1  43704         10   func_clock.html:-    func      <- start
^C
```

As each function is entered, the last columns are indented by two spaces. This shows which function is calling which: The `start()` function called `getElementById()`, which finished, and then the `start()` function called `func_a()`, and so on.

The `DELTA(us)` column shows time from that line to the previous line and so can be a bit tricky to read. The line with a delta time of 5602 us reads as "the time after `getElementById()` from line 21 finished to when `func_b()` on line 25 was executed, took 5602 us." Inspection of the code shows that these lines include a 10,000-iteration loop. Elsewhere in the code are 20,000- and 30,000-iteration loops, which can be seen in the output of `js_flowinfo.d` as taking 11197 and 16714 us, respectively. The delta time matches expectation from the code—the more iterations, the longer it takes.

The `LINE` column shows the line in the file that was being executed.

If the output looks shuffled, check the CPU `C` column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for postsorting.

See Also: js_flowtime.d

The `js_flowtime.d` script from the DTraceToolkit has similar functionality to `js_flowinfo.d` and does include a `TIME(us)` column. It is similar to the Perl version, `pl_flowtime.d`, which is demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

js_calltime.d

This script traces the time taken by JavaScript functions and object creation and prints a report. The times for functions are as follows:

Inclusive: Showing the elapsed time for functions

Exclusive: Showing which excludes time spent in other called functions

This can be used for performance analysis of JavaScript programs to identify what is causing latency.

Script

Here's the script, with the heading comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

javascript*:::function-entry
{
      self->depth++;
      self->exclude[self->depth] = 0;
      self->function[self->depth] = timestamp;
}

javascript*:::function-return
/self->function[self->depth]/
{
      this->elapsed_incl = timestamp - self->function[self->depth];
      this->elapsed_excl = this->elapsed_incl - self->exclude[self->depth];
      self->function[self->depth] = 0;
      self->exclude[self->depth] = 0;
      this->file = basename(copyinstr(arg0));
      this->name = copyinstr(arg2);

      @num[this->file, "func", this->name] = count();
      @num["-", "total", "-"] = count();
      @types_incl[this->file, "func", this->name] = sum(this->elapsed_incl);
      @types_excl[this->file, "func", this->name] = sum(this->elapsed_excl);
      @types_excl["-", "total", "-"] = sum(this->elapsed_excl);

      self->depth--;
      self->exclude[self->depth] += this->elapsed_incl;
}

javascript*:::object-create-start
{
      self->object = timestamp;
}

javascript*:::object-create-done
/self->object/
{
      this->elapsed = timestamp - self->object;
      self->object = 0;
      this->file = basename(copyinstr(arg0));
      this->name = copyinstr(arg1);

      @num[this->file, "obj-new", this->name] = count();
      @num["-", "total", "-"] = count();
      @types[this->file, "obj-new", this->name] = sum(this->elapsed);
      @types["-", "total", "-"] = sum(this->elapsed);

      self->exclude[self->depth] += this->elapsed;
}

dtrace:::END
{
      printf("\nCount,\n");
```

```
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "COUNT");
        printa("   %-20.20s %-10s %-32s %@8d\n", @num);

        normalize(@types, 1000);
        printf("\nElapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20.20s %-10s %-32s %@8d\n", @types);

        normalize(@types_excl, 1000);
        printf("\nExclusive function elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20.20s %-10s %-32s %@8d\n", @types_excl);

        normalize(@types_incl, 1000);
        printf("\nInclusive function elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20.20s %-10s %-32s %@8d\n", @types_incl);
}
```

Example

The execution of the example program `func_clock.html` was traced for three seconds:

```
# js_calltime.d
Tracing... Hit Ctrl-C to end.
^C

Count,
   FILE                 TYPE       NAME                                COUNT
   func_clock.html      func       func_a                                  3
   func_clock.html      func       func_b                                  3
   func_clock.html      func       func_c                                  3
   func_clock.html      func       setTimeout                              3
   func_clock.html      func       start                                   3
   func_clock.html      obj-new    Date                                    3
   func_clock.html      func       getElementById                         12
   -                    total      -                                      30

Elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   -                    total      -                                      15
   func_clock.html      obj-new    Date                                   15

Exclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_clock.html      func       setTimeout                            233
   func_clock.html      func       getElementById                        246
   func_clock.html      func       start                                2455
   func_clock.html      func       func_a                              16916
   func_clock.html      func       func_b                              33720
   func_clock.html      func       func_c                              49760
   -                    total      -                                  103332

Inclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_clock.html      func       setTimeout                            233
   func_clock.html      func       getElementById                        246
   func_clock.html      func       func_c                              49812
   func_clock.html      func       func_b                              83571
   func_clock.html      func       func_a                             100524
   func_clock.html      func       start                              103348
```

The difference between inclusive and exclusive function times is demonstrated by the example program: `func_a()` had 100 ms of inclusive time in total but only 17 ms of exclusive time—when its subfunction calls are excluded.

See Also: js_calldist.d

There is a variant of `js_calltime.d` in the DTraceToolkit called `js_calldist.d` (not included here), which prints times as distribution plots by subroutine name. Its functionality is similar to the Perl version, `pl_calltime.d`, which is demonstrated in the "Perl Scripts" section under `pl_callinfo.d`.

See Also: js_cputime.d, js_cpudist.d

Also in the DTraceToolkit are variants of the previous two scripts that trace on-CPU time instead of elapsed time. This time serves a different role: Elapsed time latency can include I/O wait time for system resources (network), whereas latency that is on-CPU time reflects the time to process the JavaScript code. Their functionality is similar to the Perl versions: `pl_cputime.d` is demonstrated in the "Perl Scripts" section under `pl_calltime.d`.

See Also

There are other scripts in the DTraceToolkit for tracing JavaScript in the `/JavaScript` directory. These include `js_stat.d`.

js_stat.d

This counts JavaScript events and prints per-second totals. It accepts an interval as an optional argument, similar to other `*stat` tools. Here it is tracing an idle browser that has many tabs open:

```
# js_stat.d 10
TIME                       EXEC/s   FUNC/s OBJNEW/s OBJFRE/s
2010 Jul 10 23:25:08            5      653      214        0
2010 Jul 10 23:25:18            5      779      258        0
2010 Jul 10 23:25:28            5      631      213      945
2010 Jul 10 23:25:38            5      782      259        0
2010 Jul 10 23:25:48            5      654      214        0
2010 Jul 10 23:25:58            5      766      250        0
2010 Jul 10 23:26:08            5      651      222      946
2010 Jul 10 23:26:18            5      748      248        0
2010 Jul 10 23:26:28            5      684      225        0
^C
```

Perl

The Perl programming language is a general-purpose interpreted programming language. Originally developed for text manipulation, it has undergone a massive number of enhancements, broadening its popularity and extending its use for a wide variety of programming tasks.

The scripts in this section use the perl DTrace provider,[11] which is currently not included by default in most Perl binary distributions. It has been in the Perl source since 5.10.1, which must be compiled with `Configure -Dusedtrace`. It was also available as a patch for 5.8.8, which required patching and compiling the source. Getting either of these to work requires familiarity with source compilation. Once a version of Perl that includes the perl DTrace provider has been compiled (or found), programs must be run using it for the perl provider to be visible to DTrace.

To check that the perl provider is available, attempt to list probes while a Perl program is executing:

```
# dtrace -ln 'perl*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
160934 perl117305        libperl.so                     Perl_pp_sort sub-entry
160935 perl117305        libperl.so                  Perl_pp_dbstate sub-entry
160936 perl117305        libperl.so                 Perl_pp_entersub sub-entry
160937 perl117305        libperl.so                     Perl_pp_last sub-return
160938 perl117305        libperl.so                   Perl_pp_return sub-return
160939 perl117305        libperl.so                    Perl_dounwind sub-return
160940 perl117305        libperl.so               Perl_pp_leavesublv sub-return
160941 perl117305        libperl.so                 Perl_pp_leavesub sub-return
```

This output shows that a perl provider is available, for process ID 117305, with the probes `sub-entry` and `sub-return`. The internal locations of these DTrace probes in the Perl source can be seen in the `FUNCTION` column.

Since the DTrace perl provider may be developed further, there is a chance that it has changed slightly by the time you are reading this, causing the scripts in this section to break or behave oddly. The following was the state of the provider when these scripts were written; check for changes and update the scripts accordingly:

```
provider perl {
      probe sub-entry(subroutine, file, lineno)
      probe sub-return(subroutine, file, lineno)
};
```

11. This was originally written by Alan Burlison, was later rewritten by Richard Dawe, and now is being enhanced by Sven Dowideit with extra probes included for memory allocation events.

The scripts in this section were written for and demonstrated on Perl 5.12.1 on Oracle Solaris, compiled with the fix for Perl bug #73630.[12]

Example Perl Code

The one-liners and scripts that follow are executed on the following example Perl program.

func_abc.pl

This program demonstrates function flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each function also sleeps for one second, providing known function latency that can be examined.

```
#!./perl -w

sub func_c {
    print "Function C\n";
    sleep 1;
}

sub func_b {
    print "Function B\n";
    sleep 1;
    func_c();
}

sub func_a {
    print "Function A\n";
    sleep 1;
    func_b();
}

func_a();
```

Perl One-Liners

Perl one-liners follow.

perl Provider

Trace subroutine calls:

```
dtrace -Zn 'perl*:::sub-entry { trace(copyinstr(arg0)); }'
```

12. See *http://rt.perl.org/rt3/Public/Bug/Display.html?id=73630*, which includes a fix by Peter Bray.

Count subroutine calls:

```
dtrace -Zn 'perl*:::sub-entry { @[copyinstr(arg0)] = count(); }'
```

Count subroutine calls by file:

```
dtrace -Zn 'perl*:::sub-entry { @[copyinstr(arg1)] = count(); }'
```

Perl One-Liners Selected Examples

Perl one-liner selected examples follow.

Trace Subroutine Calls

Here the one-liner traced the execution of `func_abc.pl`:

```
# dtrace -Zn 'perl*:::sub-entry { trace(copyinstr(arg0)); }'
dtrace: description 'perl*:::sub-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  6 160960       Perl_pp_entersub:sub-entry   func_a
  6 160960       Perl_pp_entersub:sub-entry   func_b
  6 160960       Perl_pp_entersub:sub-entry   func_c
^C
```

Note that the first line reads `matched 0 probes`. This was because the one-liner was executed before `func_abc.pl` or any other Perl program was running and so before there were perl probes for DTrace to see. The `-Z` option allowed this to execute; otherwise, DTrace would complain about not finding the probes.

Count Subroutine Calls by File

This time a more complex Perl program is executed, counting while file subroutines are executed from:

```
# dtrace -Zn 'perl*:::sub-entry { @[copyinstr(arg1)] = count(); }'
dtrace: description 'perl*:::sub-entry ' matched 0 probes
^C

  /opt/perl-5.12.1/lib/Carp.pm                                2
  /opt/perl-5.12.1/lib/Config_heavy.pl                        2
  /opt/perl-5.12.1/lib/warnings.pm                            2
  /opt/perl-5.12.1/lib/overload.pm                            3
  /opt/perl-5.12.1/lib/DynaLoader.pm                          4
  /opt/perl-5.12.1/lib/Exporter/Heavy.pm                      4
```

continues

```
    /opt/perl-5.12.1/lib/Time/HiRes.pm                                 4
    /opt/perl-5.12.1/lib/AutoLoader.pm                                 6
    /opt/perl-5.12.1/lib/Benchmark.pm                                  9
    /opt/perl-5.12.1/lib/warnings/register.pm                          9
    /opt/perl-5.12.1/lib/vars.pm                                      11
    /opt/perl-5.12.1/lib/Config.pm                                    12
    /opt/perl-5.12.1/lib/Exporter.pm                                  14
    /opt/perl-5.12.1/lib/constant.pm                                  21
    /opt/perl-5.12.1/lib/strict.pm                                    32
    /opt/perl-5.12.1/lib/Getopt/Long.pm                               72
    /export/home/brendan/bin/chaosreader                             375
```

The most popular file was the Perl program itself, `chaosreader`, calling 375 subroutines. The most popular library file was `Getopt/Long.pm`—the `Getopt::Long` module—which had subroutines called from it 72 times.

Perl Scripts

The scripts included in Table 8-7 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

pl_who.d

This script shows who (UID and PID) is executing which subroutines (source filename) and how many times.

Script

The `-Z` option is used so that this script can begin running before any instances of Perl—and so before there are any perl probes available to trace.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}
```

Table 8-7 Perl Script Summary

Script	Description	Provider
`pl_who.d`	Counts who is calling subroutines	perl
`pl_calls.d`	Counts subroutine calls by subroutine name	perl
`pl_flowinfo.d`	Traces subroutine flow with indented output	perl
`pl_calltime.d`	Shows inclusive and exclusive subroutine call times	perl

```

perl*:::sub-entry
{
        @lines[pid, uid, copyinstr(arg1)] = count();
}

dtrace:::END
{
        printf("   %6s %6s %6s %s\n", "PID", "UID", "SUBS", "FILE");
        printa("   %6d %6d %@6d %s\n", @lines);
}
```

Example

This has caught the execution of the three subroutines from `func_abc.pl`, showing the file path name and the user that executed it: UID 0, root.

```
# pl_who.d
Tracing... Hit Ctrl-C to end.
^C
      PID    UID   SUBS FILE
   120512      0      3 /opt/DTT/Code/Perl/func_abc.pl
```

pl_calls.d

This script counts Perl subroutine calls from any running Perl process instrumented with the perl provider.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing Perl... Hit Ctrl-C to end.\n");
}

perl*:::sub-entry
{
        @subs[pid, basename(copyinstr(arg1)), copyinstr(arg0)] = count();
}

dtrace:::END
{
        printf("%-6s %-30s %-30s %8s\n", "PID", "FILE", "SUB", "CALLS");
        printa("%-6d %-30s %-30s %@8d\n", @subs);
}
```

Examples

The following are some examples.

Known Program. The `func_abc.pl` program was traced using `pl_calls.d`, with the output matching expectations:

```
# pl_calls.d
Tracing Perl... Hit Ctrl-C to end.
^C
PID    FILE                           SUB                             CALLS
18542  func_abc.pl                    func_a                              1
18542  func_abc.pl                    func_b                              1
18542  func_abc.pl                    func_c                              1
```

The PID shows the process ID of the Perl program. `pl_calls.d` will trace all Perl programs that are running on the system, so long as the Perl versions running have the DTrace perl provider.

Complex Program. Here a much more complicated Perl program was traced, `chaosreader`, which parses and reassembles information network packet captures:

```
# pl_calls.d
Tracing Perl... Hit Ctrl-C to end.
^C
PID    FILE                           SUB                             CALLS
11854  Benchmark.pm                   clearallcache                       1
11854  Benchmark.pm                   disablecache                        1
11854  Benchmark.pm                   import                              1
11854  Benchmark.pm                   init                                1
11854  Config.pm                      AUTOLOAD                            1
11854  Config.pm                      TIEHASH                             1
11854  Config.pm                      import                              1
11854  Config_heavy.pl                BEGIN                               1
11854  Config_heavy.pl                fetch_string                        1
11854  DynaLoader.pm                  bootstrap                           1
[...output truncated...]
11854  strict.pm                      import                              9
11854  strict.pm                      unimport                           11
11854  strict.pm                      bits                               12
11854  constant.pm                    import                             13
11854  Long.pm                        BEGIN                              24
11854  Long.pm                        ParseOptionSpec                    39
11854  chaosreader                    Process_TCP_Packet                 66
11854  chaosreader                    Generate_SessionID                 70
11854  chaosreader                    Generate_IP_ID                     72
11854  chaosreader                    Read_Snoop_Record                 123
```

The output has been truncated to fit: It was a couple of pages long, showing counts of every Perl subroutine called by `chaosreader`. The most frequently called subroutine was `Read_Snoop_Record`, which was called 123 times while tracing.

See Also: pl_syscalls.d

A similar script in the DTraceToolkit is `pl_syscalls.d` (not included here), which counts subroutines and system calls from a Perl program that is either provided as `-p PID` or `-c command-to-run`:

```
# pl_syscalls.d -c /opt/DTT/Code/Perl/func_abc.pl
Tracing... Hit Ctrl-C to end.
Function A
Function B
Function C

Calls for PID 20533,

 FILE                               TYPE       NAME                        COUNT
 func_abc.pl                        sub        func_a                          1
 func_abc.pl                        sub        func_b                          1
 func_abc.pl                        sub        func_c                          1
 perl                               syscall    fcntl                           1
 perl                               syscall    getrlimit                       1
 perl                               syscall    mmap                            1
 perl                               syscall    rexit                           1
 perl                               syscall    schedctl                        1
 perl                               syscall    sigpending                      1
 perl                               syscall    sysi86                          1
 perl                               syscall    getgid                          2
 perl                               syscall    getpid                          2
 perl                               syscall    getuid                          2
 perl                               syscall    nanosleep                       3
 perl                               syscall    read                            3
 perl                               syscall    setcontext                      3
 perl                               syscall    sysconfig                       3
 perl                               syscall    write                           3
 perl                               syscall    close                           4
 perl                               syscall    llseek                          4
 perl                               syscall    open64                          4
 perl                               syscall    ioctl                           6
 perl                               syscall    gtime                           7
 perl                               syscall    brk                            24
 perl                               syscall    sigaction                      53
```

Having the Perl subroutines listed next to system calls can help explain how Perl is interacting with the operating system.

pl_flowinfo.d

This program traces Perl subroutine flow, printing various details.

Script

The `TYPE` column will only ever contain `sub`, for Perl subroutine. It's been included in case you want to enhance this script to include other event types such as libperl execution, system calls, disk I/O, and so on, which can then be examined in the context of the Perl program.

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

self int depth;

dtrace:::BEGIN
{
        printf("%s %6s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "DELTA(us)",
            "FILE", "LINE", "TYPE", "SUB");
}

perl*:::sub-entry,
perl*:::sub-return
/self->last == 0/
{
        self->last = timestamp;
}

perl*:::sub-entry
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid, this->delta,
            basename(copyinstr(arg1)), arg2, "sub", self->depth * 2, "",
            copyinstr(arg0));
        self->depth++;
        self->last = timestamp;
}

perl*:::sub-return
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%d %6d %10d  %16s:%-4d %-8s %*s<- %s\n", cpu, pid, this->delta,
            basename(copyinstr(arg1)), arg2, "sub", self->depth * 2, "",
            copyinstr(arg0));
        self->last = timestamp;
}
```

Example

Here the example program `func_abc.pl` was traced by executing it in another shell window while `pl_flowinfo.d` was running:

```
# pl_flowinfo.d
C    PID  DELTA(us)              FILE:LINE TYPE     -- SUB
0 118425          8       func_abc.pl:15   sub      -> func_a
2 118425    1000406       func_abc.pl:9    sub        -> func_b
2 118425    1000289       func_abc.pl:4    sub          -> func_c
2 118425    1000303       func_abc.pl:4    sub          <- func_c
2 118425         51       func_abc.pl:9    sub        <- func_b
2 118425         12       func_abc.pl:15   sub      <- func_a
^C
```

As each subroutine is entered, the last columns are indented by two spaces. This shows which subroutine is calling which: The previous output begins by showing that `func_a()` began and then called `func_b()`.

The DELTA(us) column shows the time from that line to the previous line, so it can be a bit tricky to read. For example, the second line of data output (skipping the header) reads as "the time from func_a() beginning to func_b() beginning was 1000406 us, or 1.00 seconds."

The LINE column shows the line in the file that was being executed.

If the output looks shuffled, check the CPU C column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for post sorting.

See Also: pl_flowtime.d

The pl_flowtime.d script (not included here) from the DTraceToolkit has similar functionality to pl_flowinfo.d and does include a TIME(us) column:

```
# pl_flowtime.d
  C TIME(us)         FILE             DELTA(us)  -- SUB
  0 883815809567     func_abc.pl              7 -> func_a
  0 883816809823     func_abc.pl        1000255   -> func_b
  0 883817810037     func_abc.pl        1000214     -> func_c
  0 883818810284     func_abc.pl        1000246     <- func_c
  0 883818810334     func_abc.pl             49   <- func_b
  0 883818810349     func_abc.pl             15 <- func_a
^C
```

The output can be sent to a file for post sorting (for example, using sort(1)), either by shell redirection > filename or by using the dtrace(1M) option -o filename (which appends, not overwrites).

See Also: pl_syscolors.d

Also in the DTraceToolkit is a variant of pl_flowinfo.d (not included here) that includes system calls in the output and uses terminal escape sequences to highlight different event types in different colors (the colors are not reproduced here):

```
# pl_syscolors.d -c func_abc.pl
C    PID  DELTA(us)              FILE:LINE TYPE     -- NAME
1 120544         10                  ":-    syscall  -> mmap
1 120544         83                  ":-    syscall  <- mmap
1 120544        175                  ":-    syscall  -> setcontext
1 120544         17                  ":-    syscall  <- setcontext
1 120544         17                  ":-    syscall  -> getrlimit
1 120544         17                  ":-    syscall  <- getrlimit
1 120544         15                  ":-    syscall  -> getpid
1 120544         11                  ":-    syscall  <- getpid
[...]
1 120547         74       func_abc.pl:15    sub      -> func_a
1 120547         87                  ":-    syscall    -> write
1 120547         87                  ":-    syscall    <- write
1 120547         32                  ":-    syscall    -> gtime
1 120547         15                  ":-    syscall    <- gtime
```

```
1 120547        26                 ":-    syscall    -> nanosleep
1 120547   1000065                 ":-    syscall    <- nanosleep
Function B
1 120547        74                 ":-    syscall    -> gtime
1 120547        36                 ":-    syscall    <- gtime
1 120547        74      func_abc.pl:9    sub        -> func_b
1 120547        70                 ":-    syscall      -> write
1 120547        86                 ":-    syscall      <- write
[...]
```

The `write()` syscalls can be seen to occur during the subroutine calls, since they wrote output to the screen. The output of the Perl program is mixed with the output of DTrace: The text `Function B` is seen, and later the output of DTrace shows the `write()` calls in `func_b()`—shown later since the DTrace output is buffered and printed later.

pl_calltime.d

This script traces the time taken by Perl subroutines (functions) to execute and prints a report. The times measured are as follows:

Inclusive: Showing the elapsed time for subroutines

Exclusive: Showing which excludes time spent in other called subroutines

This can be used for performance analysis of Perl software to identify which subroutines are responsible for latency.

Script

Here's the script, with the heading comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

perl*:::sub-entry
{
        self->depth++;
        self->exclude[self->depth] = 0;
        self->sub[self->depth] = timestamp;
}

perl*:::sub-return
/self->sub[self->depth]/
{
        this->elapsed_incl = timestamp - self->sub[self->depth];
        this->elapsed_excl = this->elapsed_incl - self->exclude[self->depth];
```

```
        self->sub[self->depth] = 0;
        self->exclude[self->depth] = 0;
        this->file = basename(copyinstr(arg1));
        this->name = copyinstr(arg0);

        @num[this->file, "sub", this->name] = count();
        @num["-", "total", "-"] = count();
        @types_incl[this->file, "sub", this->name] = sum(this->elapsed_incl);
        @types_excl[this->file, "sub", this->name] = sum(this->elapsed_excl);
        @types_excl["-", "total", "-"] = sum(this->elapsed_excl);

        self->depth--;
        self->exclude[self->depth] += this->elapsed_incl;
}

dtrace:::END
{
        printf("\nCount,\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "COUNT");
        printa("   %-20s %-10s %-32s %@8d\n", @num);

        normalize(@types_excl, 1000);
        printf("\nExclusive subroutine elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20s %-10s %-32s %@8d\n", @types_excl);

        normalize(@types_incl, 1000);
        printf("\nInclusive subroutine elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20s %-10s %-32s %@8d\n", @types_incl);
}
```

Example

The execution of the example program `func_abc.pl` was traced:

```
# pl_calltime.d
Tracing... Hit Ctrl-C to end.
^C

Count,
   FILE                 TYPE       NAME                                COUNT
   func_abc.pl          sub        func_a                                  1
   func_abc.pl          sub        func_b                                  1
   func_abc.pl          sub        func_c                                  1
   -                    total      -                                       3

Exclusive subroutine elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_abc.pl          sub        func_c                            1000215
   func_abc.pl          sub        func_a                            1000269
   func_abc.pl          sub        func_b                            1000649
   -                    total      -                                 3001135

Inclusive subroutine elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_abc.pl          sub        func_c                            1000215
   func_abc.pl          sub        func_b                            2000865
   func_abc.pl          sub        func_a                            3001135
```

The difference between inclusive and exclusive function times is demonstrated well by the example program: `func_a()` had three seconds of inclusive time but only one second of exclusive time when the subroutines that it has called are excluded.

See Also: pl_calldist.d

There is a variant of `pl_calltime.d` in the DTraceToolkit called `pl_calldist.d` (not included here), which prints times as distribution plots by subroutine name:

```
Inclusive subroutine elapsed times (us),
[...]
   chaosreader, sub, Read_Snoop_Record
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@@@@@@@@                           43
               8 |@@@@@@@@@@@@@@@@@@@@@@@@                 73
              16 |@@                                       5
              32 |@                                        2
              64 |                                         0

   chaosreader, sub, Process_TCP_Packet
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@                   37
              64 |@@@@@@@@@@@@@@@@                         26
             128 |@@                                       3
             256 |                                         0
[...]
```

This excerpt shows the `Read_Snoop_Record` subroutine was relatively fast, usually taking between 4 us and 15 us to complete.

See Also: pl_cputime.d, pl_cpudist.d

Also in the DTraceToolkit are variants of the previous two scripts that trace on-CPU time instead of elapsed time. This serves a different role: Elapsed time latency can include I/O wait time for system resources (disks, network), whereas latency that is on-CPU time is reflective of the time to process the Perl code. Modifying the script to measure on-CPU time instead of elapsed time was simply a matter of measuring deltas of the `vtimestamp` variable instead of the time stamp.

This is the `pl_cputime.d` script showing the on-CPU times of the example program:

```
# pl_cputime.d
Tracing... Hit Ctrl-C to end.
^C

Count,
   FILE                    TYPE       NAME                                   COUNT
```

```
   func_abc.pl          sub       func_a                             1
   func_abc.pl          sub       func_b                             1
   func_abc.pl          sub       func_c                             1
   -                    total     -                                  3

Exclusive subroutine on-CPU times (us),
   FILE                 TYPE      NAME                           TOTAL
   func_abc.pl          sub       func_c                           146
   func_abc.pl          sub       func_b                           184
   func_abc.pl          sub       func_a                           234
   -                    total     -                                565

Inclusive subroutine on-CPU times (us),
   FILE                 TYPE      NAME                           TOTAL
   func_abc.pl          sub       func_c                           146
   func_abc.pl          sub       func_b                           330
   func_abc.pl          sub       func_a                           565
```

Since these functions do little actual Perl code, the on-CPU times are fast—shorter than a millisecond.

PHP

PHP is a scripting language originally designed as a Web development language to produce dynamic Web pages. Web servers include a PHP processor module for executing embedded PHP code. A stand-alone interpreter enables the creation and use of scripts in PHP that can execute outside the context of a Web browser.

A PHP DTrace provider was originally developed[13] as an extension for PHP 5, which can be found on the `pecl/dtrace` module page.[14] This version involved adding a `dtrace.so` extension directive to `php.ini` and provided `function-entry` and `function-return` probes.

An enhanced PHP provider, including more probes and arguments, was later developed and added to the PHP source.[15] It was distributed as part of Oracle Sun Web Stack (previously known as Cool Stack[16]) and was recently added to the main PHP code (version PHP 5.3.99—development).[17]

To see whether your distribution of PHP has the DTrace php provider available, the compile options can be checked using `php -i` to see whether `--enable-dtrace`

13. This is by PHP core developer Wez Furlong.
14. *http://pecl.php.net/package/DTrace*
15. This is by David Soria Parra.
16. *http://hub.opensolaris.org/bin/view/Project+webstack/sunwebstack*
17. *http://blog.experimentalworks.net/2010/04/php-5-3-99-dev-and-dtrace-part-i/*

is part of the output. Another way is to run a PHP program and attempt to list probes while it is running:

```
# dtrace -ln 'php*:::'
   ID   PROVIDER         MODULE                          FUNCTION NAME
161102  php121990    mod_php5.so              dtrace_compile_file compile-file-entry
161103  php121990    mod_php5.so              dtrace_compile_file compile-file-
return
161104  php121990    mod_php5.so                       zend_error error
161105  php121990    mod_php5.so          ZEND_CATCH_SPEC_HANDLER exception-caught
161106  php121990    mod_php5.so    zend_throw_exception_internal exception-thrown
161107  php121990    mod_php5.so          dtrace_execute_internal execute-entry
161108  php121990    mod_php5.so                   dtrace_execute execute-entry
161109  php121990    mod_php5.so          dtrace_execute_internal execute-return
161110  php121990    mod_php5.so                   dtrace_execute execute-return
161111  php121990    mod_php5.so                   dtrace_execute function-entry
161112  php121990    mod_php5.so                   dtrace_execute function-return
161113  php121990    mod_php5.so      _object_and_properties_init object-create
161114  php121990    mod_php5.so      zend_objects_destroy_object object-destroy
161115  php121990    mod_php5.so             php_request_shutdown request-shutdown
[...]
```

This output shows that there is a php provider for process ID 121990 and shows the probe names in the `NAME` column. The php provider matched here is from `mod_php5.so`, and that process ID is for an Apache Web server daemon:

```
# ps -fp 121990
     UID    PID   PPID   C    STIME TTY         TIME CMD
webservd 121990 112686   0 00:20:05 ?           0:00 /usr/apache2/current/bin/httpd -f
/var/run/ak/httpd.conf -k start
```

Since the probe specification used `php*`, all Apache process IDs will be matched. This allows PHP to be traced systemwide.

The latest version of the PHP provider interface is as follows:

```
provider php {
    probe function-entry(function, file, lineno, classname, scope)
    probe function-return(function, file, lineno, classname, scope)
    probe exception-caught(classname)
    probe exception-thrown(classname)
    probe request-startup(file, uri, method)
    probe request-shutdown(file, uri, method)
    probe compile-file-entry(file, translated)
    probe compile-file-return(file, translated)
    probe error(errormsg, file, lineno)
    probe execute-entry(file, lineno)
    probe execute-return(file, lineno)
};
```

This is demonstrated in the one-liners and scripts that follow. It was executed on PHP 5.2 from Oracle Sun Web Stack (SUNWphp52r, which is now the `web/php-52` software package).

Example PHP Code

The one-liners and scripts that follow trace the following example PHP programs.

func_abc.php

This script demonstrates function flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each function also sleeps for one second, providing known function latency that can be examined.

```
<?php
function func_c()
{
      echo "Function C\n";
      sleep(1);
}

function func_b()
{
      echo "Function B\n";
      sleep(1);
      func_c();
}

function func_a()
{
      echo "Function A\n";
      sleep(1);
      func_b();
}

func_a();
?>
```

broken.php

Here's `broken.php`:

```
<?php
echo "Example PHP program with error\n";
bogus text here
echo "Done\n";
?>
```

PHP One-Liners

PHP one-liners are presented in this section.

php Provider

Trace function calls showing function name:

```
dtrace -Zn 'php*:::function-entry { trace(copyinstr(arg0)); }'
```

Trace program execution filename:

```
dtrace -Zn 'php*:::request-startup { trace(copyinstr(arg0)); }'
```

Count function calls by function name:

```
dtrace -Zn 'php*:::function-entry { @[copyinstr(arg0)] = count(); }'
```

Count function calls by filename:

```
dtrace -Zn 'php*:::function-entry { @[copyinstr(arg1)] = count(); }'
```

Count program execution by filename:

```
dtrace -Zn 'php*:::request-startup { @[copyinstr(arg0)] = count(); }'
```

Count line execution by filename and line number:

```
dtrace -Zn 'php*:::execute-entry { @[copyinstr(arg0), arg1] = count(); }'
```

Trace PHP errors:

```
dtrace -Zn 'php*:::error { printf("%s:%d: \"%s\"", copyinstr(arg1), arg2,
 copyinstr(arg0)); }'
```

Count all PHP events

```
dtrace -Zn 'php*::: { @[probename] = count(); }'
```

PHP One-Liners Selected Examples

PHP one-liner selected examples are presented in this section.

Trace Function Calls Showing Function Name

The execution of `func_abc.php` is traced using this one-liner.

```
# dtrace -Zn 'php*:::function-entry { trace(copyinstr(arg0)); }'
dtrace: description 'php*:::function-entry ' matched 18 probes
CPU     ID                    FUNCTION:NAME
  3  96371    dtrace_execute:function-entry   func_a
  3  96371    dtrace_execute:function-entry   func_b
  3  96371    dtrace_execute:function-entry   func_c
^C
```

Count Function Calls by Filename

Here we loaded a Web site that uses the MediaWiki wiki software,[18] which is PHP based:

```
# dtrace -Zn 'php*:::function-entry { @[copyinstr(arg1)] = count(); }'
dtrace: description 'php*:::function-entry ' matched 15 probes
^C

  /var/htdocs/wiki/includes/normal/UtfNormal.php               1
  /var/htdocs/wiki/StartProfiler.php                        2
  /var/htdocs/wiki/includes/DefaultSettings.php               2
  /var/htdocs/wiki/LocalSettings.php                        4
[...]
  /var/htdocs/wiki/index.php                               46
  /var/htdocs/wiki/includes/GlobalFunctions.php              66
  /var/htdocs/wiki/includes/BagOStuff.php                  72
  /var/htdocs/wiki/includes/User.php                       81
  /var/htdocs/wiki/includes/Wiki.php                       92
  /var/htdocs/wiki/includes/StubObject.php                131
  /var/htdocs/wiki/includes/IP.php                        140
  /var/htdocs/wiki/includes/AutoLoader.php                148
  /var/htdocs/wiki/includes/Setup.php                     190
  /var/htdocs/wiki/includes/Database.php                  214
  /var/htdocs/wiki/includes/LoadBalancer.php              248
  /var/htdocs/wiki/includes/MessageCache.php              372
  /var/htdocs/wiki/languages/Language.php                 390
  /var/htdocs/wiki/includes/Parser.php                    473
  /var/htdocs/wiki/includes/MagicWord.php                 549
```

18. *www.mediawiki.org/wiki/MediaWiki*

The output shows the source files and function counts while a single page was loaded. The includes/MagicWord.php file was the source for 549 function calls. The names of these functions can be examined via the arg0 variable.

Trace PHP Errors

The example broken.php program was executed while tracing for errors:

```
# dtrace -Zn 'php*:::error { printf("%s:%d: \"%s\"", copyinstr(arg1), arg2,
copyinstr(arg0)); }'
dtrace: description 'php*:::error ' matched 19 probes
CPU     ID                    FUNCTION:NAME
  7  96190  zend_error:error /var/htdocs/wiki/broken.php:3: "syntax error, unexpected
T_STRING"
^C
```

This has correctly identified the file, line number, and type of error. Similar capability can be found by searching for PHP errors in the Web server log file; an advantage of the DTrace probe is that it could be included as part of a larger script, perhaps recording the function flow that led to the error.

PHP Scripts

The scripts included in Table 8-8 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

php_calls.d

This script counts PHP function calls from any PHP program on the system instrumented with the php provider.

Script

Instead of displaying the full path name to the file, it is processed by basename() to remove the directory component.

Table 8-8 PHP Script Summary

Script	Description	Provider
php_calls.d	Counts function calls by function name	php
php_flowinfo.d	Traces function flow with indented output	php

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing PHP... Hit Ctrl-C to end.\n");
}

php*:::function-entry
{
        @funcs[basename(copyinstr(arg1)), copyinstr(arg0)] = count();
}

dtrace:::END
{
        printf(" %-32s %-32s %8s\n", "FILE", "FUNC", "CALLS");
        printa(" %-32s %-32s %@8d\n", @funcs);
}
```

Examples

Here are some examples.

Example Program. The `func_abc.php` program was traced using `php_calls.d`, which showed the expected number of function calls:

```
# php_calls.d
Tracing PHP... Hit Ctrl-C to end.
^C
 FILE                             FUNC                                CALLS
 func_abc.php                     func_a                                  1
 func_abc.php                     func_b                                  1
 func_abc.php                     func_c                                  1
```

MediaWiki. This script traces a MediaWiki page loading, and the output has been truncated to fit. All of the functions called and their source files can be seen, with the execution count.

```
# php_calls.d
Tracing PHP... Hit Ctrl-C to end.
^C
 FILE                             FUNC                                CALLS
 Article.php                      addGoodLinkObj                          1
 Article.php                      checkLastModified                       1
 Article.php                      checkTouched                            1
 Article.php                      getArticleID                            1
[...]
 LoadBalancer.php                 isOpen                                255
 Language.php                     isMultibyte                           261
 MessageCache.php                 wfProfileIn                           338
 MessageCache.php                 wfProfileOut                          338
 MagicWord.php                    getMagic                              345
```

continues

```
MagicWord.php                     __construct                          350
MagicWord.php                     load                                 350
Parser.php                        get                                  372
```

php_flowinfo.d

This program traces PHP function flow, showing time stamps.

Script

This script was written for an earlier version of the PHP provider, which sometimes passed a NULL pointer as the function name. Lines 69 and 80 check that the function name pointer is valid (not NULL).

```
 1  #!/usr/sbin/dtrace -Zs
[...]
50  #pragma D option quiet
51  #pragma D option switchrate=10
52
53  self int depth;
54
55  dtrace:::BEGIN
56  {
57          printf("%s %6s/%-4s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "TID",
58              "DELTA(us)", "FILE", "LINE", "TYPE", "FUNC");
59  }
60
61  php*:::function-entry,
62  php*:::function-return
63  /self->last == 0/
64  {
65          self->last = timestamp;
66  }
67
68  php*:::function-entry
69  /arg0/
70  {
71          this->delta = (timestamp - self->last) / 1000;
72          printf("%d %6d/%-4d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid, tid,
73              this->delta, basename(copyinstr(arg1)), arg2, "func",
74              self->depth * 2, "", copyinstr(arg0));
75          self->depth++;
76          self->last = timestamp;
77  }
78
79  php*:::function-return
80  /arg0/
81  {
82          this->delta = (timestamp - self->last) / 1000;
83          self->depth -= self->depth > 0 ? 1 : 0;
84          printf("%d %6d/%-4d %10d  %16s:%-4d %-8s %*s<- %s\n", cpu, pid, tid,
85              this->delta, basename(copyinstr(arg1)), arg2, "func",
86              self->depth * 2, "", copyinstr(arg0));
87          self->last = timestamp;
88  }
```

The TYPE column will only contain func for PHP function; this script can be enhanced to include other types, such as php library internal functions, system calls, and so on.

Example

The func_abc.php program was traced using php_flowinfo.d:

```
# php_flowinfo.d
C    PID/TID   DELTA(us)              FILE:LINE TYPE     -- FUNC
7 122231/1            10      func_abc.php:22   func     -> func_a
7 122231/1       1000145      func_abc.php:19   func       -> func_b
7 122231/1       1000140      func_abc.php:12   func         -> func_c
7 122231/1       1000111      func_abc.php:12   func         <- func_c
7 122231/1            56      func_abc.php:19   func       <- func_b
7 122231/1            15      func_abc.php:22   func     <- func_a
^C
```

As each function is entered, the last columns are indented by two spaces. This shows which function is calling which: the previous output begins by showing that func_a() began and then called func_b().

The DELTA(us) column shows time from that line to the previous line. This shows the sleep commands are taking around 1.01 seconds, as expected.

If the output looks shuffled, check the CPU C column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for postsorting.

See Also: php_flowtime.d, php_syscolors.d

The php_flowtime.d script from the DTraceToolkit is similar to php_flowinfo.d, including a time stamp column that can be postsorted if the output is shuffled. Another similar script from the DTraceToolkit is php_syscolors.d, which includes system calls in the output and uses terminal escape sequences to highlight different event types in different colors. They are similar to the Perl versions, pl_flowtime.d and pl_syscolors.d, which are demonstrated in the "Perl Scripts" section under pl_flowinfo.d.

See Also

The DTraceToolkit has other scripts for the php provider, including php_calltime.d for a report of inclusive and exclusive function time, and variants. See the Perl versions of these scripts in the "Perl Scripts" section for similar example output.

Python

The Python programming language is a general-purpose, interpreted language that was built around code readability through clean syntax, offering the programmer choices in terms of which method of development they prefer (object-oriented, procedural, and so on). Python is therefore often described as a multiparadigm language, because it supports several different programming paradigms.

The scripts in this section use the python DTrace provider.[19] Patches for different Python versions are available on the Python bugs page for "Issue 4111: Add Systemtap/DTrace probes,"[20] which requires familiarity with source compilation to get working. Some distributions already have these built in, such as Python 2.6.4 on OpenSolaris. Once a version of Python including the python DTrace provider has been found (or compiled), programs must be run using it for the provider to be visible to DTrace.

To check that the python provider is available, attempt to list probes while a Python program is executing:

```
# dtrace -ln 'python*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
160958 python120694 libpython2.6.so.1.0           PyEval_EvalFrameEx function-entry
160959 python120694 libpython2.6.so.1.0                 dtrace_entry function-entry
160960 python120694 libpython2.6.so.1.0           PyEval_EvalFrameEx function-return
160961 python120694 libpython2.6.so.1.0                dtrace_return function-return
```

This output shows that there is a python provider for process ID 120694, with the probes `function-entry` and `function-return`.

Since the DTrace Python provider may be developed further, there is a chance that it has changed slightly by the time you are reading this, causing these scripts in this section to break or behave oddly. The following was the state of the provider when these scripts were written—check for changes and update the scripts accordingly:

```
provider python {
    probe function-entry(file, subroutine, lineno)
    probe function-return(file, subroutine, lineno)
};
```

The scripts in this section were written for and demonstrated on Python 2.6.4 on Oracle Solaris, which includes the python provider.

19. This was originally written by John Levon.

20. *http://bugs.python.org/issue4111*

Example Python Code

The one-liners and scripts that follow trace the following example Python program.

func_abc.py

This program demonstrates function flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each function also sleeps for one second, providing known function latency that can be examined.

```
#!/usr/bin/python

import time

def func_c():
      print "Function C"
      time.sleep(1)

def func_b():
      print "Function B"
      time.sleep(1)
      func_c()

def func_a():
      print "Function A"
      time.sleep(1)
      func_b()

func_a()
```

Python One-Liners

Python one-liners are presented in this section.

python Provider

Trace function calls:

```
dtrace -Zn 'python*:::function-entry { trace(copyinstr(arg1)); }'
```

Count function calls:

```
dtrace -Zn 'python*:::function-entry { @[copyinstr(arg1)] = count(); }'
```

Count subroutine calls by file:

```
dtrace -Zn 'python*:::function-entry { @[copyinstr(arg0)] = count(); }'
```

Profile Python stack traces at 123 Hertz:

```
dtrace -n 'profile-123 /pid == $target/ { @[jstack()] = count(); }' -p PID
```

Python One-Liners Selected Examples

Python one-liner selected examples are presented in this section.

Trace Function Calls

The execution of `func_abc.py` is traced using this one-liner:

```
# dtrace -Zn 'python*:::function-entry { trace(copyinstr(arg1)); }'
dtrace: description 'python*:::function-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  7 160959      dtrace_entry:function-entry   <module>
[...]
  7 160959      dtrace_entry:function-entry   Codec
  7 160959      dtrace_entry:function-entry   IncrementalEncoder
  7 160959      dtrace_entry:function-entry   IncrementalDecoder
  7 160959      dtrace_entry:function-entry   StreamWriter
  7 160959      dtrace_entry:function-entry   StreamReader
  7 160959      dtrace_entry:function-entry   StreamConverter
  7 160959      dtrace_entry:function-entry   getregentry
  7 160959      dtrace_entry:function-entry   __new__
  7 160959      dtrace_entry:function-entry   <module>
  7 160959      dtrace_entry:function-entry   func_a
  7 160959      dtrace_entry:function-entry   func_b
  7 160959      dtrace_entry:function-entry   func_c
^C
```

Note that the first line reads `matched 0 probes`. This was because the one-liner was executed before `func_abc.py` or any other Python program was running and so before there were `python` probes for DTrace to see. The `-Z` option allowed this to execute; otherwise, DTrace would complain about not finding the probes.

Count Function Calls by File

```
# dtrace -Zn 'python*:::function-entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'python*:::function-entry ' matched 0 probes
^C

  /usr/lib/python2.6/encodings/aliases.py                          1
  /usr/lib/python2.6/linecache.py                                  1
  /usr/lib/python2.6/types.py                                      3
  /opt/DTT/Code/Python/func_abc.py                                 4
  /usr/lib/python2.6/encodings/__init__.py                         4
  /usr/lib/python2.6/warnings.py                                   5
```

```
   /usr/lib/python2.6/copy_reg.py                                    7
   /usr/lib/python2.6/encodings/ascii.py                             8
   /usr/lib/python2.6/UserDict.py                                    9
   /usr/lib/python2.6/genericpath.py                                11
   <string>                                                         11
   /usr/lib/python2.6/codecs.py                                     12
   /usr/lib/python2.6/os.py                                         14
   /usr/lib/python2.6/stat.py                                       15
   /usr/lib/python2.6/_abcoll.py                                    31
   /usr/lib/python2.6/site.py                                       41
   /usr/lib/python2.6/abc.py                                        85
   /usr/lib/python2.6/posixpath.py                                 119
```

The most popular file was `posixpath.py`, from which 119 functions were called. The `func_abc.py` program is in the output, with four functions called: the three from the program and the import of the time module.

Profile Python Stack Traces

The python provider also enhances the DTrace `jstack()` action to incorporate Python functions into the user stack trace. Here the stack trace was sampled at 123 Hertz:

```
# dtrace -n 'profile-123 /pid == $target/ { @[jstack()] = count(); }'
-c ./func_slow.py
dtrace: description 'profile-123 ' matched 1 probe
[...]

              libpython2.6.so.1.0`PyEval_EvalFrameEx+0x2da
                [ ./func_slow.py:3 (func_c) ]
              libpython2.6.so.1.0`fast_function+0x108
              libpython2.6.so.1.0`call_function+0xee
              libpython2.6.so.1.0`PyEval_EvalFrameEx+0x3029
                [ ./func_slow.py:16 (func_b) ]
              libpython2.6.so.1.0`fast_function+0x108
              libpython2.6.so.1.0`call_function+0xee
              libpython2.6.so.1.0`PyEval_EvalFrameEx+0x3029
                [ ./func_slow.py:24 (func_a) ]
              libpython2.6.so.1.0`fast_function+0x108
              libpython2.6.so.1.0`call_function+0xee
              libpython2.6.so.1.0`PyEval_EvalFrameEx+0x3029
                [ ./func_slow.py:26 (<module>) ]
              libpython2.6.so.1.0`PyEval_EvalCodeEx+0x91c
              libpython2.6.so.1.0`PyEval_EvalCode+0x32
              libpython2.6.so.1.0`run_mod+0x3a
              libpython2.6.so.1.0`PyRun_FileExFlags+0x6b
              libpython2.6.so.1.0`PyRun_SimpleFileExFlags+0x189
              libpython2.6.so.1.0`PyRun_AnyFileExFlags+0x6e
              libpython2.6.so.1.0`Py_Main+0xa94
              isapython2.6`main+0x63
              isapython2.6`_start+0x7d
                6
```

The Python insertions have been highlighted in this stack trace. This provides a remarkable insight into how Python code is executed internally by Python.

Table 8-9 Python Script Summary

Script	Description	Provider
`py_who.d`	Counts who is calling functions	python
`py_calls.d`	Counts function calls by function name	python
`py_flowinfo.d`	Traces function flow with indented output	python
`py_calltime.d`	Shows inclusive and exclusive function call times	python

Python Scripts

The scripts included in Table 8-9 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

py_who.d

This script shows who (UID and PID) is executing which functions (source filename) and how many times.

Script

The `-Z` option is used so that this script can begin running before any instances of Python and so before there are any `python` probes available to trace.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

python*:::function-entry
{
        @lines[pid, uid, copyinstr(arg0)] = count();
}

dtrace:::END
{
        printf("   %6s %6s %6s %s\n", "PID", "UID", "FUNCS", "FILE");
        printa("   %6d %6d %@6d %s\n", @lines);
}
```

Example

This has caught the execution of four functions from `func_abc.py`, showing the file path name (it's shipped in the DTraceToolkit) and the user who executed it: UID 0, root.

```
# py_who.d
Tracing... Hit Ctrl-C to end.
^C
      PID    UID  FUNCS FILE
   120704      0      1 /usr/lib/python2.6/encodings/aliases.py
   120704      0      1 /usr/lib/python2.6/linecache.py
   120704      0      3 /usr/lib/python2.6/types.py
   120704      0      4 /opt/DTT/Code/Python/func_abc.py
   120704      0      4 /usr/lib/python2.6/encodings/__init__.py
   120704      0      5 /usr/lib/python2.6/warnings.py
   120704      0      7 /usr/lib/python2.6/copy_reg.py
   120704      0      8 /usr/lib/python2.6/encodings/ascii.py
   120704      0      9 /usr/lib/python2.6/UserDict.py
   120704      0     11 /usr/lib/python2.6/genericpath.py
   120704      0     11 <string>
   120704      0     12 /usr/lib/python2.6/codecs.py
   120704      0     14 /usr/lib/python2.6/os.py
   120704      0     15 /usr/lib/python2.6/stat.py
   120704      0     31 /usr/lib/python2.6/_abcoll.py
   120704      0     41 /usr/lib/python2.6/site.py
   120704      0     85 /usr/lib/python2.6/abc.py
   120704      0    119 /usr/lib/python2.6/posixpath.py
```

py_calls.d

This script counts Python function calls from any running python process instrumented with the python provider.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing Python... Hit Ctrl-C to end.\n");
}

python*:::function-entry
{
        @funcs[pid, basename(copyinstr(arg0)), copyinstr(arg1)] = count();
}

dtrace:::END
{
        printf("%-6s %-30s %-30s %8s\n", "PID", "FILE", "FUNC", "CALLS");
        printa("%-6d %-30s %-30s %@8d\n", @funcs);
}
```

Example

The `func_abc.py` program was traced using `py_calls.d`:

```
# py_calls.d
Tracing Python... Hit Ctrl-C to end.
^C
PID    FILE                          FUNC                                CALLS
120731 UserDict.py                   <module>                                1
120731 UserDict.py                   DictMixin                               1
120731 UserDict.py                   IterableUserDict                        1
[...]
120731 stat.py                       S_ISDIR                                 7
120731 _abcoll.py                    __subclasshook__                       10
120731 abc.py                        __subclasscheck__                      10
120731 abc.py                        register                               10
120731 genericpath.py                isdir                                  10
120731 os.py                         _exists                                10
120731 <string>                      <module>                               11
120731 posixpath.py                  normcase                               14
120731 site.py                       makepath                               14
120731 abc.py                        abstractmethod                         15
120731 abc.py                        __new__                                16
120731 posixpath.py                  join                                   20
120731 posixpath.py                  abspath                                27
120731 posixpath.py                  isabs                                  27
120731 posixpath.py                  normpath                               27
120731 abc.py                        <genexpr>                              31
```

The PID shows the process ID of the Python program. `py_calls.d` will trace all Python programs that are running on the system, so long as the python versions running have the DTrace python provider.

py_flowinfo.d

This program traces Python function flow, printing various details.

Script

The `TYPE` column will only ever contain `func`, for Python functions. You can enhance this script to include other event types such as libpython execution, system calls, disk I/O, and so on, which can then be examined in the context of the Python program.

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

self int depth;

dtrace:::BEGIN
{
        printf("%s %6s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "DELTA(us)",
            "FILE", "LINE", "TYPE", "FUNC");
}

python*:::function-entry,
python*:::function-return
```

```
/self->last == 0/
{
        self->last = timestamp;
}

python*:::function-entry
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid, this->delta,
            basename(copyinstr(arg0)), arg2, "func", self->depth * 2, "",
            copyinstr(arg1));
        self->depth++;
        self->last = timestamp;
}

python*:::function-return
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%d %6d %10d  %16s:%-4d %-8s %*s<- %s\n", cpu, pid, this->delta,
            basename(copyinstr(arg0)), arg2, "func", self->depth * 2, "",
            copyinstr(arg1));
        self->last = timestamp;
}
```

Example

While tracing, the `func_abc.py` program was executed in another shell window:

```
# py_flowinfo.d
C    PID  DELTA(us)              FILE:LINE TYPE     -- FUNC
4 120737          8           site.py:59   func     -> <module>
4 120737        952             os.py:22   func       -> <module>
4 120737       1086      posixpath.py:11   func         -> <module>
4 120737        325           stat.py:4    func           -> <module>
4 120737         45           stat.py:94   func           <- <module>
4 120737        247    genericpath.py:5    func           -> <module>
[...]
4 120737         45          ascii.py:41   func       <- <module>
4 120737         60          ascii.py:41   func       -> getregentry
4 120737         28         codecs.py:77   func         -> __new__
4 120737         24         codecs.py:87   func         <- __new__
4 120737         21          ascii.py:49   func       <- getregentry
4 120737         33      __init__.py:154   func     <- search_function
4 120737        879      func_abc.py:3     func     -> <module>
4 120737       1917      func_abc.py:14    func       -> func_a
4 120737    1000293      func_abc.py:9     func         -> func_b
4 120737    1000211      func_abc.py:5     func           -> func_c
4 120737    1000223      func_abc.py:7     func           <- func_c
4 120737         62      func_abc.py:12    func         <- func_b
4 120737         14      func_abc.py:17    func       <- func_a
4 120737         13      func_abc.py:19    func     <- <module>
^C
```

The output of `py_flowinfo.d` has been truncated to fit. The functions called (and modules loaded) when initializing Python have been traced, followed by the execution of the program.

As each function is entered, the last columns are indented by two spaces. This shows which function is calling which: The previous output begins by showing that `func_a()` began and then called `func_b()`.

The `DELTA(us)` column shows time from that line to the previous line so can be a bit tricky to read. For example, the line showing a time of 1000293 us reads as "the time from `func_a()` beginning to `func_b()` beginning was 1000293 us, or 1.00 seconds."

The `LINE` column shows the line in the file what was being executed.

If the output looks shuffled, check the CPU `C` column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for postsorting.

See Also: py_flowtime.d

The `py_flowtime.d` script from the DTraceToolkit has similar functionality to `py_flowinfo.d` and does include a `TIME(us)` column. Its functionality is similar to the Perl version, `pl_flowtime.d`, which is demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

See Also: py_syscolors.d

Also in the DTraceToolkit is a variant called `py_syscolors.d` that includes system calls in the output and uses terminal escape sequences to highlight different event types in different colors. Its functionality is similar to the Perl version, `pl_syscolors.d`, which is demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

py_calltime.d

This script traces the time taken by Python functions to execute and prints a report. The times measured are as follows:

- **Inclusive**: Showing the elapsed time for subroutines
- **Exclusive**: Showing which excludes time spent in other called subroutines

This can be used for performance analysis of Perl software to identify which subroutines are responsible for latency.

Script

Here's the script, with the heading comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

python*:::function-entry
{
      self->depth++;
      self->exclude[self->depth] = 0;
      self->function[self->depth] = timestamp;
}

python*:::function-return
/self->function[self->depth]/
{
      this->elapsed_incl = timestamp - self->function[self->depth];
      this->elapsed_excl = this->elapsed_incl - self->exclude[self->depth];
      self->function[self->depth] = 0;
      self->exclude[self->depth] = 0;
      this->file = basename(copyinstr(arg0));
      this->name = copyinstr(arg1);

      @num[this->file, "func", this->name] = count();
      @num["-", "total", "-"] = count();
      @types_incl[this->file, "func", this->name] = sum(this->elapsed_incl);
      @types_excl[this->file, "func", this->name] = sum(this->elapsed_excl);
      @types_excl["-", "total", "-"] = sum(this->elapsed_excl);

      self->depth--;
      self->exclude[self->depth] += this->elapsed_incl;
}

dtrace:::END
{
      printf("\nCount,\n");
      printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "COUNT");
      printa("   %-20s %-10s %-32s %@8d\n", @num);

      normalize(@types_excl, 1000);
      printf("\nExclusive function elapsed times (us),\n");
      printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
      printa("   %-20s %-10s %-32s %@8d\n", @types_excl);

      normalize(@types_incl, 1000);
      printf("\nInclusive function elapsed times (us),\n");
      printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
      printa("   %-20s %-10s %-32s %@8d\n", @types_incl);
}
```

Example

The execution of the example program `func_abc.py` was traced:

```
# py_calltime.d
Tracing... Hit Ctrl-C to end.
^C
```

continues

```
Count,
   FILE                 TYPE       NAME                                COUNT
   UserDict.py          func       <module>                                1
[...]
   posixpath.py         func       abspath                                27
   posixpath.py         func       isabs                                  27
   posixpath.py         func       normpath                               27
   abc.py               func       <genexpr>                              31
   -                    total      -                                     381

Exclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   site.py              func       setencoding                             3
[...]
   os.py                func       _exists                              1756
   __init__.py          func       <module>                             1855
   abc.py               func       __new__                              2079
   func_abc.py          func       <module>                             2109
   os.py                func       <module>                             2376
   func_abc.py          func       func_c                            1000214
   func_abc.py          func       func_a                            1000292
   func_abc.py          func       func_b                            1000683
   -                    total      -                                 3032947

Inclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   site.py              func       setencoding                             3
[...]
   UserDict.py          func       <module>                             7071
   site.py              func       main                                 9115
   os.py                func       <module>                            15402
   site.py              func       <module>                            25720
   func_abc.py          func       func_c                            1000214
   func_abc.py          func       func_b                            2000898
   func_abc.py          func       func_a                            3001191
   func_abc.py          func       <module>                          3003301
```

The output has been truncated to fit. The difference between inclusive and exclusive function times is demonstrated well by the example program: `func_a()` had three seconds of inclusive time but only one second of exclusive time—when its subfunction calls are excluded.

See Also py_calldist.d

There is a variant of `pl_calltime.d` in the DTraceToolkit called `py_calldist.d` (not included here), which prints times as distribution plots by function name. Its functionality is similar to the Perl version, `pl_calltime.d`, which is demonstrated in the "Perl Scripts" section under `pl_callinfo.d`.

See Also: py_cputime.d, py_cpudist.d

Also in the DTraceToolkit are variants of the previous two scripts that trace on-CPU time instead of elapsed time. This serves a different role: Elapsed time latency can include I/O wait time for system resources (disks, network), whereas latency that is on-CPU time reflects the time to process the Python code. Their functionality is similar to the Perl versions: `pl_cputime.d` is demonstrated in the "Perl Scripts" section under `pl_calltime.d`.

Ruby

Ruby is a general-purpose, interpreted, object-oriented programming language. Like Python, Ruby supports multiple programming paradigms and was designed for programming productivity and code readability.

These scripts trace activity of the Ruby programming language and require the DTrace ruby provider.[21] The ruby provider was made available as a separate download either in patch, source, or binary form from the "Ruby DTrace" page on Joyent.[22] The Ruby distribution shipped with Mac OS X Leopard integrated the Joyent ruby provider, and a similar version is available in MacRuby.[23] There is also a Ruby interface for DTrace available,[24] `ruby-dtrace`, allowing DTrace scripts to be executed from a Ruby program.

To check that the DTrace ruby provider is available, attempt to list probes while a Ruby program is executing:

```
# dtrace -ln 'ruby*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
20406  ruby11649   libruby.1.dylib                          rb_call0 function-entry
20407  ruby11649   libruby.1.dylib                          rb_call0 function-return
20408  ruby11649   libruby.1.dylib                   garbage_collect gc-begin
20409  ruby11649   libruby.1.dylib                   garbage_collect gc-end
20410  ruby11649   libruby.1.dylib                           rb_eval line
20411  ruby11649   libruby.1.dylib                      rb_obj_alloc object-create-done
20412  ruby11649   libruby.1.dylib                      rb_obj_alloc object-create-start
20413  ruby11649   libruby.1.dylib                   garbage_collect object-free
20414  ruby11649   libruby.1.dylib                        rb_longjmp raise
20415  ruby11649   libruby.1.dylib                           rb_eval rescue
20416  ruby11649   libruby.1.dylib                 ruby_dtrace_probe ruby-probe
```

This output shows that there is a ruby provider for process ID 11649 and shows the probe names (`NAME` column) along with their location in the Ruby source (`FUNCTION` column).

21. This was written by Scott Barron of Joyent.

22. *https://dtrace.joyent.com/projects/ruby-dtrace/wiki/Ruby+DTrace*

23. *www.macruby.org/trac/wiki/WhatsNewInLeopard*

24. *http://ruby-dtrace.rubyforge.org/*

The DTrace Ruby provider interface is as follows:

```
provider ruby {
    probe function-entry(class, method, file, lineno);
    probe function-return(class, method, file, lineno);
    probe raise(errinfo, file, lineno);
    probe rescue(file, lineno);
    probe line(file, lineno);
    probe gc-begin();
    probe gc-end();
    probe object-create-start(object, file, lineno);
    probe object-create-done(object, file, lineno);
    probe object-free(object);
};
```

Note that while Ruby calls its functions `methods`, the provider traces them with `function-entry` and `function-return` probes. If this interface has changed for the ruby provider version you are using, update the one-liners and scripts that follow accordingly.

This section demonstrates the ruby provider as shipped in ruby 1.8.7 on Mac OS X version 10.6.

Example Ruby Code

The one-liners and scripts that follow trace the following example Ruby program.

func_abc.rb

This program demonstrates method flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each method also sleeps for one second, providing known method latency that can be examined.

```
#!/usr/bin/ruby -w

def func_c
  print "Function C\n"
  sleep 1
end

def func_b
  print "Function B\n"
  sleep 1
  func_c
end

def func_a
  print "Function A\n"
  sleep 1
  func_b
end

func_a
```

Ruby One-Liners

Ruby one-liners follow.

ruby Provider

Trace method calls showing class and method:

```
dtrace -Zn 'ruby*:::function-entry { printf("%s::%s", copyinstr(arg0),
copyinstr(arg1)); }'
```

Count method calls by method name:

```
dtrace -Zn 'ruby*:::function-entry { @[copyinstr(arg1)] = count(); }'
```

Count method calls by filename:

```
dtrace -Zn 'ruby*:::function-entry { @[copyinstr(arg2)] = count(); }'
```

Count line execution by filename and line number:

```
dtrace -Zn 'ruby*:::line { @[basename(copyinstr(arg0)), arg1] = count(); }'
```

Count object creation by object class name:

```
dtrace -Zn 'ruby*:::object-create-done { @[copyinstr(arg0)] = count(); }'
```

Trace garbage collection events with nanosecond time stamps:

```
dtrace -Zn 'ruby*:::gc-* { trace(timestamp); }'
```

Ruby One-Liners Selected Examples

Ruby one-liner selected examples follow.

Trace Method Calls Showing Class and Method

The execution of `func_abc.rb` is traced using this one-liner, which prints class and method names separated by a period:

```
# dtrace -Zn 'ruby*:::function-entry { printf("%s.%s", copyinstr(arg0),
copyinstr(arg1)); }'
dtrace: description 'ruby*:::function-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  0  73153          rb_call0:function-entry Module.method_added
  0  73153          rb_call0:function-entry Module.method_added
  0  73153          rb_call0:function-entry Module.method_added
  0  73153          rb_call0:function-entry Object.func_a
  0  73153          rb_call0:function-entry Object.print
  0  73153          rb_call0:function-entry IO.write
  0  73153          rb_call0:function-entry Object.sleep
  0  73153          rb_call0:function-entry Object.func_b
  0  73153          rb_call0:function-entry Object.print
  0  73153          rb_call0:function-entry IO.write
  0  73153          rb_call0:function-entry Object.sleep
  0  73153          rb_call0:function-entry Object.func_c
  0  73153          rb_call0:function-entry Object.print
  0  73153          rb_call0:function-entry IO.write
  0  73153          rb_call0:function-entry Object.sleep
^C
```

Note that the first line reads `matched 0 probes`. This was because the one-liner was executed before `func_abc.rb` or any other Ruby program was running and so before there were ruby probes for DTrace to see. The `-Z` option allowed this to execute; otherwise, DTrace would complain about not finding the probes.

All methods can be seen in the output: the methods from `func_abc.rb`, as well as calls to IO.write to write the output text.

Count Method Calls by Filename

The filename from which the methods are called is traced by this one-liner:

```
# dtrace -Zn 'ruby*:::function-entry { @[copyinstr(arg2)] = count(); }'
dtrace: description 'ruby*:::function-entry ' matched 0 probes
^C

  /opt/DTT/Code/Ruby/func_abc.rb                                   15
```

All methods were invoked from `func_abc.rb` program.

Count Line Execution by Filename and Line Number

This one-liner uses the line probe to trace filename and line number:

```
# dtrace -Zn 'ruby*:::line { @[basename(copyinstr(arg0)), arg1] = count(); }'
dtrace: description 'ruby*:::line ' matched 0 probes
^C

  func_slow.rb                                                  3            1
  func_slow.rb                                                  4            1
  func_slow.rb                                                  5            1
  func_slow.rb                                                  6            1
[...]
  func_slow.rb                                                 26       100000
  func_slow.rb                                                 27       100000
  func_slow.rb                                                 16       200000
  func_slow.rb                                                 17       200000
  func_slow.rb                                                  7       300000
  func_slow.rb                                                  8       300000
```

The output indicates that the `func_slow.rb` program (not included in this book) executed lines 7 and 8 some 300,000 times each. This matches the source, which executed those lines in a loop:

```
     6    while i < 300000
     7       i = i + 1
     8       j = i + 1
     9    end
```

Ruby Scripts

The scripts included in Table 8-10 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

rb_who.d

This script shows who (UID and PID) is executing how many lines of Ruby from which filename.

Table 8-10 Ruby Script Summary

Script	Description	Provider
`rb_who.d`	Counts who is calling methods	ruby
`rb_calls.d`	Counts method calls by method name	ruby
`rb_flowinfo.d`	Traces method flow with indented output	ruby
`rb_calltime.d`	Shows inclusive and exclusive method call times	ruby

Script

The -Z option is used so that this script can begin running before any instances of Ruby, so before there are any ruby probes available to trace.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

ruby*:::line
{
      @lines[pid, uid, copyinstr(arg0)] = count();
}

dtrace:::END
{
      printf("   %6s %6s %10s %s\n", "PID", "UID", "LINES", "FILE");
      printa("   %6d %6d %@10d %s\n", @lines);
}
```

Example

While tracing, the func_abc.rb program was executed in another shell window, which was found to run 12 lines of Ruby:

```
# rb_who.d
Tracing... Hit Ctrl-C to end.
^C
      PID    UID      LINES FILE
    11711    501         12 /opt/DTT/Code/Ruby/func_abc.rb
```

rb_calls.d

This script counts Ruby method calls from any running ruby process instrumented with the ruby provider.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing Ruby... Hit Ctrl-C to end.\n");
}

```

```
ruby*:::function-entry
{
        @funcs[pid, basename(copyinstr(arg2)), copyinstr(arg0),
          copyinstr(arg1)] = count();
}

dtrace:::END
{
        printf("%-6s %-28.28s %-16s %-16s %8s\n", "PID", "FILE", "CLASS",
          "METHOD", "CALLS");
        printa("%-6d %-28.28s %-16s %-16s %@8d\n", @funcs);
}
```

Example

The `func_abc.rb` program was traced using `rb_calls.d`:

```
# rb_calls.d
Tracing Ruby... Hit Ctrl-C to end.
^C
PID    FILE                          CLASS            METHOD              CALLS
11722  func_abc.rb                   Object           func_a                  1
11722  func_abc.rb                   Object           func_b                  1
11722  func_abc.rb                   Object           func_c                  1
11722  func_abc.rb                   IO               write                   3
11722  func_abc.rb                   Module           method_added            3
11722  func_abc.rb                   Object           print                   3
11722  func_abc.rb                   Object           sleep                   3
```

The PID shows the process ID of the Ruby program. `rb_calls.d` will trace all Ruby programs that are running on the system that have the DTrace ruby provider.

rb_flowinfo.d

This program traces Ruby method flow, printing various details.

Script

The `TYPE` column will only ever contain `method` for Ruby methods. This script can be enhanced to include other event types such as libruby execution, system calls, disk I/O, and so on, which can then be examined in the context of the Ruby program.

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

self int depth;

dtrace:::BEGIN
{
```

continues

```
            printf("%s %6s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "DELTA(us)",
                "FILE", "LINE", "TYPE", "NAME");
      }

      ruby*:::function-entry,
      ruby*:::function-return
      /self->last == 0/
      {
            self->last = timestamp;
      }

      ruby*:::function-entry
      {
            this->delta = (timestamp - self->last) / 1000;
            this->name = strjoin(strjoin(copyinstr(arg0), "::"), copyinstr(arg1));
            printf("%d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid, this->delta,
                basename(copyinstr(arg2)), arg3, "method", self->depth * 2, "",
                this->name);
            self->depth++;
            self->last = timestamp;
      }

      ruby*:::function-return
      {
            this->delta = (timestamp - self->last) / 1000;
            self->depth -= self->depth > 0 ? 1 : 0;
            this->name = strjoin(strjoin(copyinstr(arg0), "::"), copyinstr(arg1));
            printf("%d %6d %10d  %16s:%-4d %-8s %*s<- %s\n", cpu, pid, this->delta,
                basename(copyinstr(arg2)), arg3, "method", self->depth * 2, "",
                this->name);
            self->last = timestamp;
      }
```

Example

The `func_abc.rb` program was traced using rb_`flowinfo.d`:

```
# rb_flowinfo.d
C    PID  DELTA(us)              FILE:LINE TYPE     -- NAME
0  11801          2       func_abc.rb:3    method   -> Module::method_added
0  11801         41       func_abc.rb:3    method   <- Module::method_added
0  11801         37       func_abc.rb:8    method   -> Module::method_added
0  11801         25       func_abc.rb:8    method   <- Module::method_added
0  11801         29       func_abc.rb:14   method   -> Module::method_added
0  11801         25       func_abc.rb:14   method   <- Module::method_added
0  11801         33       func_abc.rb:20   method   -> Object::func_a
0  11801         24       func_abc.rb:15   method     -> Object::print
0  11801         24       func_abc.rb:15   method       -> IO::write
0  11801        164       func_abc.rb:15   method       <- IO::write
0  11801         23       func_abc.rb:15   method     <- Object::print
0  11801         24       func_abc.rb:16   method     -> Object::sleep
0  11801    1000074       func_abc.rb:16   method     <- Object::sleep
0  11801         44       func_abc.rb:17   method     -> Object::func_b
0  11801         33       func_abc.rb:9    method       -> Object::print
0  11801         24       func_abc.rb:9    method         -> IO::write
0  11801         28       func_abc.rb:9    method         <- IO::write
0  11801         21       func_abc.rb:9    method       <- Object::print
0  11801         23       func_abc.rb:10   method       -> Object::sleep
0  11801    1000062       func_abc.rb:10   method       <- Object::sleep
0  11801         45       func_abc.rb:11   method       -> Object::func_c
0  11801         32       func_abc.rb:4    method         -> Object::print
```

```
  0  11801         46       func_abc.rb:4    method           -> IO::write
  0  11801         28       func_abc.rb:4    method           <- IO::write
  0  11801         21       func_abc.rb:4    method         <- Object::print
  0  11801         23       func_abc.rb:5    method         -> Object::sleep
  0  11801    1000063       func_abc.rb:5    method         <- Object::sleep
  0  11801         38       func_abc.rb:5    method       <- Object::func_c
  0  11801         24       func_abc.rb:11   method     <- Object::func_b
  0  11801         24       func_abc.rb:17   method   <- Object::func_a
  ^C
```

The output of `py_flowinfo.d` is truncated to fit. The functions called (and modules loaded) when initializing Ruby have been traced, followed by the execution of the program.

As each function is entered, the last columns are indented by two spaces. This shows which function is calling which: The previous output begins by showing that `func_a()` began and then called `func_b()`.

The `DELTA(us)` column shows time from that line to the previous line and so can be a bit tricky to read. This example is particularly easy because it has traced the entry to return of the `sleep()` methods, each taking about 1.00 seconds.

If the output looks shuffled, check the CPU `C` column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for postsorting.

See Also: rb_flowtime.d

The `rb_flowtime.d` script from the DTraceToolkit has similar functionality to `rb_flowinfo.d` and does include a `TIME(us)` column. It is similar to the Perl version, `pl_flowtime.d`, which is demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

See Also: rb_syscolors.d

Also in the DTraceToolkit is a variant called `rb_syscolors.d`, which includes system calls in the output and uses terminal escape sequences to highlight different event types in different colors. It is similar to the Perl version, `pl_syscolors.d`, which is demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

rb_calltime.d

This script traces the time taken by Ruby methods, object creation, and garbage collection and prints a report. The times for methods are as follows:

Inclusive: Showing the elapsed time for methods

Exclusive: Showing which excludes time spent in other called methods

This can be used for performance analysis of Ruby software to identify what is responsible for latency.

Script

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

ruby*:::function-entry
{
      self->depth++;
      self->exclude[self->depth] = 0;
      self->function[self->depth] = timestamp;
}

ruby*:::function-return
/self->function[self->depth]/
{
      this->elapsed_incl = timestamp - self->function[self->depth];
      this->elapsed_excl = this->elapsed_incl - self->exclude[self->depth];
      self->function[self->depth] = 0;
      self->exclude[self->depth] = 0;
      this->file = basename(copyinstr(arg2));
      this->name = strjoin(strjoin(copyinstr(arg0), "::"), copyinstr(arg1));

      @num[this->file, "func", this->name] = count();
      @num["-", "total", "-"] = count();
      @types_incl[this->file, "func", this->name] = sum(this->elapsed_incl);
      @types_excl[this->file, "func", this->name] = sum(this->elapsed_excl);
      @types_excl["-", "total", "-"] = sum(this->elapsed_excl);

      self->depth--;
      self->exclude[self->depth] += this->elapsed_incl;
}

ruby*:::object-create-start
{
      self->object = timestamp;
}

ruby*:::object-create-done
/self->object/
{
      this->elapsed = timestamp - self->object;
      self->object = 0;
      this->file = basename(copyinstr(arg1));
      this->file = this->file != NULL ? this->file : ".";
      this->name = copyinstr(arg0);

      @num[this->file, "obj-new", this->name] = count();
      @types[this->file, "obj-new", this->name] = sum(this->elapsed);

      self->exclude[self->depth] += this->elapsed;
}

```

```
ruby*:::gc-begin
{
        self->gc = timestamp;
}

ruby*:::gc-end
/self->gc/
{
        this->elapsed = timestamp - self->gc;
        self->gc = 0;
        @num[".", "gc", "-"] = count();
        @types[".", "gc", "-"] = sum(this->elapsed);
        self->exclude[self->depth] += this->elapsed;
}

dtrace:::END
{
        printf("\nCount,\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "COUNT");
        printa("   %-20s %-10s %-32s %@8d\n", @num);

        normalize(@types, 1000);
        printf("\nElapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20s %-10s %-32s %@8d\n", @types);

        normalize(@types_excl, 1000);
        printf("\nExclusive function elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20s %-10s %-32s %@8d\n", @types_excl);

        normalize(@types_incl, 1000);
        printf("\nInclusive function elapsed times (us),\n");
        printf("   %-20s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "TOTAL");
        printa("   %-20s %-10s %-32s %@8d\n", @types_incl);
}
```

Example

The execution of the example program `func_abc.rb` was traced:

```
# rb_calltime.d
Tracing... Hit Ctrl-C to end.
^C

Count,
   FILE                 TYPE       NAME                                COUNT
   .                    obj-new    NoMemoryError                           1
   .                    obj-new    SystemStackError                        1
   .                    obj-new    ThreadGroup                             1
   .                    obj-new    fatal                                   1
   func_abc.rb          func       Object::func_a                          1
   func_abc.rb          func       Object::func_b                          1
   func_abc.rb          func       Object::func_c                          1
   .                    obj-new    Object                                  3
   func_abc.rb          func       IO::write                               3
   func_abc.rb          func       Module::method_added                    3
   func_abc.rb          func       Object::print                           3
   func_abc.rb          func       Object::sleep                           3
   -                    total      -                                      15
```

continues

```
Elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   .                    obj-new    SystemStackError                        4
   .                    obj-new    fatal                                   9
   .                    obj-new    NoMemoryError                           9
   .                    obj-new    ThreadGroup                            11
   .                    obj-new    Object                                 27

Exclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_abc.rb          func       Module::method_added                   10
   func_abc.rb          func       IO::write                             115
   func_abc.rb          func       Object::func_c                        392
   func_abc.rb          func       Object::func_b                        444
   func_abc.rb          func       Object::print                         473
   func_abc.rb          func       Object::func_a                        521
   func_abc.rb          func       Object::sleep                     3000324
   -                    total      -                                 3002281

Inclusive function elapsed times (us),
   FILE                 TYPE       NAME                                TOTAL
   func_abc.rb          func       Module::method_added                   10
   func_abc.rb          func       IO::write                             115
   func_abc.rb          func       Object::print                         588
   func_abc.rb          func       Object::func_c                    1000523
   func_abc.rb          func       Object::func_b                    2001179
   func_abc.rb          func       Object::sleep                     3000324
   func_abc.rb          func       Object::func_a                    3002271
```

The output has been truncated to fit. The difference between inclusive and exclusive function times is demonstrated well by the example program: `func_a()` had three seconds of inclusive time but only one second of exclusive time—when its subfunction calls are excluded.

See Also: rb_calldist.d

There is a variant of `rb_calltime.d` in the DTraceToolkit called `rb_calldist.d` (not included here), which prints times as distribution plots by subroutine name. Its functionality is similar to the Perl version, `pl_calltime.d`, which is demonstrated in the "Perl Scripts" section under `pl_callinfo.d`.

See Also: rb_cputime.d, rb_cpudist.d

Also in the DTraceToolkit are variants of the previous two scripts that trace on-CPU time instead of elapsed time. This serves a different role: elapsed time latency can include I/O wait time for system resources (disks, network), whereas latency that is on-CPU time is reflective of the time to process the Ruby code. Their functionality is similar to the Perl versions: `pl_cputime.d` is demonstrated in the "Perl Scripts" section under `pl_calltime.d`.

See Also

Other Ruby scripts are in the DTraceToolkit in the `/Ruby` directory.

rb_stat.d

This counts Ruby events from all running Ruby software on the system and prints per-second totals. It accepts an interval as an optional argument, similar to other `*stat` tools.

```
# rb_stat.d
TIME                       EXEC/s METHOD/s OBJNEW/s OBJFRE/s RAIS/s RESC/s   GC/s
2010 Jul  9 22:41:32            0        0        0        0      0      0      0
2010 Jul  9 22:41:33            1    90550        7        0      0      0      0
2010 Jul  9 22:41:34            0   551264        0        0      0      0      0
2010 Jul  9 22:41:35            0   556786        0        0      0      0      0
2010 Jul  9 22:41:36            0   559991        0        0      0      0      0
2010 Jul  9 22:41:37            0    41419        0        0      0      0      0
2010 Jul  9 22:41:38            0        0        0        0      0      0      0
^C
```

rb_malloc.d

This script uses the pid provider in addition to the ruby provider to trace libc `malloc()` calls on Oracle Solaris so that memory allocations can be seen in the context of Ruby code:

```
# rb_malloc.d -c ./func_abc.rb
Tracing... Hit Ctrl-C to end.
Function A
Function B
Function C
Ruby malloc byte distributions by recent Ruby operation,
[...]
   func_abc.rb, method, IO::write
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
            8192 |                                         0

   ., objnew, SystemStackError
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         3
               4 |@@@@                                     32
               8 |@@                                       15
              16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          279
              32 |@@@                                      30
              64 |                                         2
             128 |                                         0
[...]
```

The script can be modified to run on other operating systems by adjusting the probe name for the `malloc()` routine.

Shell

A DTrace provider for the Bourne shell (`sh`) has been written,[25] which can be used for the analysis of shell script operation. Provider documentation, source patch, and binary versions of `sh` for x86 and SPARC are available on the DTrace Providers for Various Shells site.[26] It is not yet shipped by default on any operating system.

To check that the sh provider is available, try to list probes while a Bourne shell is running:

```
# dtrace -ln 'sh*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
160958   sh121038                sh                            execute builtin-entry
160959   sh121038                sh                            execute builtin-return
160960   sh121038                sh                            execute command-entry
160961   sh121038                sh                            execute command-return
160962   sh121038                sh                            execute function-entry
160963   sh121038                sh                            execute function-return
160964   sh121038                sh                            execute line
160965   sh121038                sh                             exfile script-done
160966   sh121038                sh                             exitsh script-done
160967   sh121038                sh                             exfile script-start
160968   sh121038                sh                            execute subshell-entry
160969   sh121038                sh                            execute subshell-return
```

This output shows that there is a sh provider for process ID 121038 and shows the probe names in the `NAME` column. The Bourne shell provider interface is described by PSARC 2008/008[27] and is as follows:

```
provider sh {
    probe function-entry(file, function, lineno);
    probe function-return(file, function, rval);
    probe builtin-entry(file, function, lineno);
    probe builtin-return(file, function, rval);
    probe command-entry(file, function, lineno);
    probe command-return(file, function, rval);
    probe script-start(file);
    probe script-done(file, rval);
    probe subshell-entry(file, childpid);
    probe subshell-return(file, rval);
    probe line(file, lineno);
    probe variable-set(file, variable, value);
    probe variable-unset(file, variable);
};
```

25. This is by Alan Hargreaves.

26. *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/shells*

27. *http://arc.opensolaris.org/caselog/PSARC/2008/008/*

This section demonstrates the sh provider on Oracle Solaris, using the `sh` executable from the Providers for Various Shells site.

Example Shell Code

The one-liners and scripts that follow trace this example shell script.

func_abc.sh

This script demonstrates function flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each function also sleeps for one second, providing known function latency that can be examined.

```
#!/usr/bin/sh

func_c()
{
        echo "Function C"
        sleep 1
}

func_b()
{
        echo "Function B"
        sleep 1
        func_c
}

func_a()
{
        echo "Function A"
        sleep 1
        func_b
}

func_a
```

Shell One-Liners

Shell one-liners follow.

sh Provider

Trace function calls showing function name:

```
dtrace -Zn 'sh*:::function-entry { trace(copyinstr(arg1)); }'
```

Trace command execution showing command name:

```
dtrace -Zn 'sh*:::command-entry { trace(copyinstr(arg1)); }'
```

Trace built-in calls showing builtin name:

```
dtrace -Zn 'sh*:::builtin-entry { trace(copyinstr(arg1)); }'
```

Count function calls by function name:

```
dtrace -Zn 'sh*:::function-entry { @[copyinstr(arg1)] = count(); }'
```

Count function calls by filename:

```
dtrace -Zn 'sh*:::function-entry { @[copyinstr(arg0)] = count(); }'
```

Count line execution by filename and line number:

```
dtrace -Zn 'sh*:::line { @[basename(copyinstr(arg0)), arg1] = count(); }'
```

Shell One-Liners Selected Examples

Shell one-liner selected examples follow.

Trace Function Calls Showing Function Name

The execution of `func_abc.sh` is traced using this one-liner.

```
# dtrace -Zn 'sh*:::function-entry { trace(copyinstr(arg1)); }'
dtrace: description 'sh*:::function-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  6 160962           execute:function-entry   func_a
  6 160962           execute:function-entry   func_b
  6 160962           execute:function-entry   func_c
^C
```

Note that the first line reads `matched 0 probes`. This was because the one-liner was executed before `func_abc.sh` or any other (instrumented) Bourne shell was running and so before there were sh probes for DTrace to see. The -Z option allowed this to execute; otherwise, DTrace would complain about not finding the probes.

Count Function Calls by Filename

All three functions are invoked from the `func_abc.sh` script.

```
# dtrace -Zn 'sh*:::function-entry { @[copyinstr(arg0)] = count(); }'
dtrace: description 'sh*:::function-entry ' matched 0 probes
^C

  /opt/DTT/Code/Shell/func_abc.sh                                   3
```

Count Line Execution by Filename and Line Number

This one-liner uses the line probe to trace the filename and line number:

```
# dtrace -Zn 'sh*:::line { @[basename(copyinstr(arg0)), arg1] = count(); }'
dtrace: description 'sh*:::line ' matched 0 probes
^C

  func_slow.sh                                                     5                1
[...]
  func_slow.sh                                                    19              200
  func_slow.sh                                                    17              201
  func_slow.sh                                                     9              300
  func_slow.sh                                                     7              301
  func_slow.sh                                                     1              600
```

The output indicates that the `func_slow.sh` program (not included in this book) executed line 7 exactly 301 times and line 9 exactly 300 times. This matches the source, which executed those lines in a loop:

```
 6          i=0
 7          while [ $i -lt 300 ]
 8          do
 9                  i=`expr $i + 1`
10          done
```

The output also indicated that line 1 was executed 600 times; this is not line 1 of the shell script, which is the interpreter line but is line 1 of shell code run in command substitution subshells (the code between ` on line 9). This could be differentiated a number of ways, including using the `subshell-entry` and `subshell-return` probes or by including the PID in the output.

Shell Scripts

The scripts included in Table 8-11 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

sh_who.d

This script shows who (UID and PID) is executing how many lines of shell from which filename.

Script

The `-Z` option is used so that this script can begin running before any instances of the instrumented Bourne shell—and so before there are any `sh` probes available to trace.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

sh*:::line
{
        @lines[pid, uid, copyinstr(arg0)] = count();
}

dtrace:::END
{
        printf("   %6s %6s %6s %s\n", "PID", "UID", "LINES", "FILE");
        printa("   %6d %6d %@6d %s\n", @lines);
}
```

Examples

Example `sh_who.d` scripts follow.

Table 8-11 Shell Script Summary

Script	Description	Provider
`sh_who.d`	Counts who is calling how many lines of shell	sh
`sh_calls.d`	Counts function/builtin/command calls	sh
`sh_flowinfo.d`	Traces function flow with indented output	sh

Example Script. While tracing, the `func_abc.sh` program was executed in another shell window:

```
# sh_who.d
Tracing... Hit Ctrl-C to end.
^C
      PID    UID  LINES FILE
   121791      0      9 /opt/DTT/Code/Shell/func_abc.sh
```

This has traced `func_abc.sh` executing nine lines of shell.

Production Script. Here we trace an instance of starting Mozilla Firefox:

```
# sh_who.d
Tracing... Hit Ctrl-C to end.
^C
      PID    UID  LINES FILE
    13678    100      1 firefox
    13679    100      1 firefox
    13680    100      1 firefox
    13681    100      1 firefox
    13683    100      1 firefox
    13685    100      1 firefox
    13686    100      1 firefox
    13687    100      1 firefox
    13690    100      1 firefox
    13693    100      1 /usr/lib/firefox/run-mozilla.sh
    13694    100      1 /usr/lib/firefox/run-mozilla.sh
    13695    100      1 /usr/lib/firefox/run-mozilla.sh
    13692    100     55 /usr/lib/firefox/run-mozilla.sh
    13677    100     75 firefox
```

Firefox itself (PID 13677) ran 75 lines of code. There are also instances of `firefox` running a single line of code with a different PID each time, which are calls to subshells.

sh_calls.d

This script counts shell function and built-in and command calls from any running Bourne shells on the system that are instrumented with the sh provider.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

```

continues

```
dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

sh*:::function-entry
{
        @calls[basename(copyinstr(arg0)), "func", copyinstr(arg1)] = count();
}

sh*:::builtin-entry
{
        @calls[basename(copyinstr(arg0)), "builtin", copyinstr(arg1)] = count();
}

sh*:::command-entry
{
        @calls[basename(copyinstr(arg0)), "cmd", copyinstr(arg1)] = count();
}

sh*:::subshell-entry
/arg1 != 0/
{
        @calls[basename(copyinstr(arg0)), "subsh", "-"] = count();
}

dtrace:::END
{
        printf(" %-22s %-10s %-32s %8s\n", "FILE", "TYPE", "NAME", "COUNT");
        printa(" %-22s %-10s %-32s %@8d\n", @calls);
}
```

Examples

Example scripts follow.

Example Script. The `func_abc.sh` script was traced using `sh_calls.d`:

```
# sh_calls.d
Tracing... Hit Ctrl-C to end.
^C
 FILE                   TYPE       NAME                                COUNT
 func_abc.sh            func       func_a                                  1
 func_abc.sh            func       func_b                                  1
 func_abc.sh            func       func_c                                  1
 func_abc.sh            builtin    echo                                    3
 func_abc.sh            cmd        sleep                                   3
```

The three functions are visible, along with three calls to the `echo` shell built in and three calls to the `sleep(1)` command.

Production Script. The following traced the Mozilla Firefox start script:

```
# sh_calls.d
Tracing... Hit Ctrl-C to end.
^C
 FILE                   TYPE       NAME                                COUNT
 firefox                builtin    .                                       1
 firefox                builtin    break                                   1
 firefox                builtin    exit                                    1
 firefox                builtin    pwd                                     1
 firefox                builtin    test                                    1
 firefox                cmd        /usr/lib/firefox/run-mozilla.sh         1
 run-mozilla.sh         builtin    break                                   1
 run-mozilla.sh         builtin    exit                                    1
 run-mozilla.sh         builtin    return                                  1
 run-mozilla.sh         builtin    shift                                   1
 run-mozilla.sh         builtin    type                                    1
 run-mozilla.sh         cmd        /usr/lib/firefox/firefox-bin            1
 run-mozilla.sh         func       moz_run_program                         1
 run-mozilla.sh         func       moz_test_binary                         1
 firefox                builtin    echo                                    2
 firefox                func       moz_pis_startstop_scripts               2
 firefox                builtin    cd                                      3
 firefox                builtin    export                                  3
 run-mozilla.sh         builtin    export                                  3
 firefox                builtin    :                                       6
 firefox                func       moz_spc_verbose_echo                    6
 run-mozilla.sh         subsh      -                                       9
 firefox                builtin    [                                      18
 firefox                subsh      -                                      20
 run-mozilla.sh         builtin    [                                      20
```

The Firefox start script called `run-mozilla.sh`, which can be seen both as a `cmd` call in the previous output from the `firefox` script and as additional calls from the `run-mozilla.sh` script.

The built-in called `[` is the test built-in and was called 20 times by `run-mozilla.sh` and 18 times by `firefox`. The `firefox` script also called 20 subshells.

sh_flowinfo.d

This program traces Shell function flow, printing various details.

Script

Here's the script, with header comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]

#pragma D option quiet
#pragma D option switchrate=10

self int depth;

```

continues

```
dtrace:::BEGIN
{
        self->depth = 0;
        printf("%3s %6s %10s  %16s:%-4s %-8s -- %s\n", "C", "PID", "DELTA(us)",
            "FILE", "LINE", "TYPE", "NAME");
}

sh*:::function-entry,
sh*:::function-return,
sh*:::builtin-entry,
sh*:::builtin-return,
sh*:::command-entry,
sh*:::command-return,
sh*:::subshell-entry,
sh*:::subshell-return
/self->last == 0/
{
        self->last = timestamp;
}

sh*:::function-entry
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%3d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), arg2, "func",
            self->depth * 2, "", copyinstr(arg1));
        self->depth++;
        self->last = timestamp;
}

sh*:::function-return
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%3d %6d %10d  %16s:-    %-8s %*s<- %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), "func", self->depth * 2,
            "", copyinstr(arg1));
        self->last = timestamp;
}

sh*:::builtin-entry
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%3d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), arg2, "builtin",
            self->depth * 2, "", copyinstr(arg1));
        self->depth++;
        self->last = timestamp;
}

sh*:::builtin-return
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%3d %6d %10d  %16s:-    %-8s %*s<- %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), "builtin",
            self->depth * 2, "", copyinstr(arg1));
        self->last = timestamp;
}

sh*:::command-entry
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%3d %6d %10d  %16s:%-4d %-8s %*s-> %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), arg2, "cmd",
```

```
            self->depth * 2, "", copyinstr(arg1));
        self->depth++;
        self->last = timestamp;
}

sh*:::command-return
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%3d %6d %10d  %16s:-    %-8s %*s<- %s\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), "cmd",
            self->depth * 2, "", copyinstr(arg1));
        self->last = timestamp;
}

sh*:::subshell-entry
/arg1 != 0/
{
        this->delta = (timestamp - self->last) / 1000;
        printf("%3d %6d %10d  %16s:-    %-8s %*s-> pid %d\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), "subsh",
            self->depth * 2, "", arg1);
        self->depth++;
        self->last = timestamp;
}

sh*:::subshell-return
/self->last/
{
        this->delta = (timestamp - self->last) / 1000;
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%3d %6d %10d  %16s:-    %-8s %*s<- = %d\n", cpu, pid,
            this->delta, basename(copyinstr(arg0)), "subsh",
            self->depth * 2, "", arg1);
        self->last = timestamp;
}
```

Example

The `func_abc.sh` program was traced using `sh_flowinfo.d`:

```
# sh_flowinfo.d
  C    PID  DELTA(us)              FILE:LINE TYPE     -- NAME
  0 121880          7       func_abc.sh:23   func     -> func_a
  0 121880         72       func_abc.sh:18   builtin    -> echo
  0 121880        109       func_abc.sh:-    builtin    <- echo
  0 121880       8997       func_abc.sh:19   cmd        -> sleep
  0 121880    1012848       func_abc.sh:-    cmd        <- sleep
  0 121880        113       func_abc.sh:20   func       -> func_b
  0 121880         48       func_abc.sh:11   builtin      -> echo
  0 121880         96       func_abc.sh:-    builtin      <- echo
  0 121880       8486       func_abc.sh:12   cmd          -> sleep
  0 121880    1014084       func_abc.sh:-    cmd          <- sleep
  0 121880        118       func_abc.sh:13   func         -> func_c
  0 121880         48       func_abc.sh:5    builtin        -> echo
  0 121880         94       func_abc.sh:-    builtin        <- echo
  0 121880       7852       func_abc.sh:6    cmd            -> sleep
  0 121880    1012783       func_abc.sh:-    cmd            <- sleep
  0 121880         91       func_abc.sh:-    func         <- func_c
  0 121880         46       func_abc.sh:-    func       <- func_b
  0 121880         10       func_abc.sh:-    func     <- func_a
^C
```

As each function is entered, the last columns are indented by two spaces. This shows which function is calling which: The previous output begins by showing that `func_a()` began and then called `func_b()`.

The `DELTA(us)` column shows time from that line to the previous line. This shows that the sleep commands are taking around 1.01 seconds, as expected.

If the output looks shuffled, check the CPU `C` column—the output can shuffle when the CPU ID changes from one line to the next. If this becomes a problem, a time stamp column can be included in the output for postsorting.

See Also: sh_flowtime.d, sh_syscolors.d

The `sh_flowtime.d` script from the DTraceToolkit has similar functionality to `sh_flowinfo.d` and does include a `TIME(us)` column. Another similar script is `sh_syscolors`, which includes system calls in the output, highlighted in different colors using terminal escape sequences. They are similar to the Perl versions, `pl_flowtime.d` and `pl_syscolors.d`, which are demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

See Also

The DTraceToolkit has other scripts for the sh provider, including `sh_calltime.d` for a report of inclusive and exclusive function time, and variants. Their functionality is similar to the Perl versions, which can be seen under `pl_calltime.d` in the "Perl Scripts" section.

Tcl

Tcl (often pronounced "tickle") is a scripting language, so TCL programs are executed under an interpreter. Tcl evolved as a popular language to enable rapid software prototyping, including GUI development with the use of the tk GUI toolkit, where it is often used to add value to other applications.

These scripts trace activity of the Tcl programming language, making use of the Tcl DTrace provider,[28] which was integrated into the Tcl source in version tcl8.4.16 and 8.5b1. See "DTrace" on the Tcl wiki for details.[29]

28. This was written by Daniel Steffen.
29. *http://wiki.tcl.tk/19923*

For the Tcl DTrace provider to be available, the Tcl source must be compiled with the `--enable-dtrace` option. Getting this working requires familiarity with source compilation.

To check that the tcl provider is available, attempt to list probes while a Tcl program is executing:

```
# dtrace -ln 'tcl*:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
63285     tcl807      libtcl8.4.so              TclEvalObjvInternal cmd-args
63286     tcl807      libtcl8.4.so              TclEvalObjvInternal cmd-entry
63287     tcl807      libtcl8.4.so              TclEvalObjvInternal cmd-result
63288     tcl807      libtcl8.4.so              TclEvalObjvInternal cmd-return
63289     tcl807      libtcl8.4.so               TclExecuteByteCode inst-done
63290     tcl807      libtcl8.4.so               TclExecuteByteCode inst-start
63291     tcl807      libtcl8.4.so                     TclPtrSetVar obj-create
63292     tcl807      libtcl8.4.so                    Tcl_ConcatObj obj-create
[...output truncated...]
63343     tcl807      libtcl8.4.so               CallCommandTraces obj-free
63344     tcl807      libtcl8.4.so                TclRenameCommand obj-free
63345     tcl807      libtcl8.4.so                TclObjInterpProc proc-args
63346     tcl807      libtcl8.4.so                TclObjInterpProc proc-entry
63347     tcl807      libtcl8.4.so                TclObjInterpProc proc-result
63348     tcl807      libtcl8.4.so                TclObjInterpProc proc-return
63349     tcl807      libtcl8.4.so                    DTraceObjCmd tcl-probe
```

This output shows that there is a tcl provider for process ID 807 and shows the probe names in the `NAME` column. The DTrace Tcl provider interface is described on the wiki and in the source file `generic/tclDTrace.d`. It is as follows:

```
provider tcl {
    probe proc-entry(procname, argc, argv);
    probe proc-return(procname, retcode);
    probe proc-result(procname, retcode, retval, retobj);
    probe proc-args(procname, args, ...);
    probe cmd-entry(cmdname, argc, argv);
    probe cmd-return(cmdname, retval);
    probe cmd-result(cmdname, retcode, retval, retobj);
    probe cmd-args(procname, args, ...);
    probe inst-start(instname, depth, stackobj);
    probe inst-done(instname, depth, stackobj);
    probe obj-create(object);
    probe obj-free(object);
    probe tcl-probe(strings, ...);
};
```

This section demonstrates the Tcl provider on Oracle Solaris, using Tcl 8.4.16.

Example Tcl Code

The one-liners and scripts that follow trace the following example Tcl program.

func_abc.tcl

This script demonstrates procedure flow: `func_a()` calls `func_b()`, which calls `func_c()`. Each procedure also sleeps for one second, providing known procedure latency that can be examined.

```
#!./tclsh

proc func_c {} {
      puts "Function C"
      after 1000
}

proc func_b {} {
      puts "Function B"
      after 1000
      func_c
}

proc func_a {} {
      puts "Function A"
      after 1000
      func_b
}

func_a
```

Tcl One-Liners

Tcl one-liners follow.

tcl Provider

Trace procedure calls showing procedure name:

```
dtrace -Zn 'tcl*:::proc-entry { trace(copyinstr(arg0)); }'
```

Trace command calls showing command name:

```
dtrace -Zn 'tcl*:::cmd-entry { trace(copyinstr(arg0)); }'
```

Count procedure calls by procedure name:

```
dtrace -Zn 'tcl*:::proc-entry { @[copyinstr(arg0)] = count(); }'
```

Count command calls by command name:

```
dtrace -Zn 'tcl*:::proc-entry { @[copyinstr(arg0)] = count(); }'
```

Count object allocation by object name:

```
dtrace -Zn 'tcl*:::object-create { @[copyinstr(arg0)] = count(); }'
```

Count all Tcl events:

```
dtrace -Zn 'tcl*::: { @[probename] = count(); }'
```

Tcl One-Liners Selected Examples

Tcl one-liner selected example follow.

Trace Procedure Calls Showing Procedure Name

The execution of `func_abc.sh` is traced using this one-liner.

```
# dtrace -Zn 'tcl*:::proc-entry { trace(copyinstr(arg0)); }'
dtrace: description 'tcl*:::proc-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  0  63346      TclObjInterpProc:proc-entry   tclInit
  0  63346      TclObjInterpProc:proc-entry   func_a
  0  63346      TclObjInterpProc:proc-entry   func_b
  0  63346      TclObjInterpProc:proc-entry   func_c
^C
```

Note that the first line reads `matched 0 probes`. This was because the one-liner was executed before `func_abc.tcl` or any other Tcl program was running and so before there were tcl probes for DTrace to see. The `-Z` option allowed this to execute; otherwise, DTrace would complain about not finding the probes.

The first function was `tclInit()`, which is part of Tcl initialization for program execution.

Table 8-12 Tcl Script Summary

Script	Description	Provider
tcl_who.d	Counts who is calling how many Tcl commands	tcl
tcl_calls.d	Counts procedure and command calls	tcl
tcl_procflow.d	Traces procedure flow with indented output	tcl

Trace Command Calls Showing Command Name

This is tracing `func_abc.tcl` and shows the execution of all the built-in Tcl commands, as well as the procedure calls.

```
# dtrace -Zn 'tcl*:::cmd-entry { trace(copyinstr(arg0)); }'
dtrace: description 'tcl*:::cmd-entry ' matched 0 probes
CPU     ID                    FUNCTION:NAME
  1  63286    TclEvalObjvInternal:cmd-entry   if
  1  63286    TclEvalObjvInternal:cmd-entry   info
  1  63286    TclEvalObjvInternal:cmd-entry   proc
  1  63286    TclEvalObjvInternal:cmd-entry   tclInit
[...output truncated...]
  1  63286    TclEvalObjvInternal:cmd-entry   func_a
  1  63286    TclEvalObjvInternal:cmd-entry   puts
  1  63286    TclEvalObjvInternal:cmd-entry   after
  0  63286    TclEvalObjvInternal:cmd-entry   func_b
  0  63286    TclEvalObjvInternal:cmd-entry   puts
  0  63286    TclEvalObjvInternal:cmd-entry   after
  0  63286    TclEvalObjvInternal:cmd-entry   func_c
  0  63286    TclEvalObjvInternal:cmd-entry   puts
  0  63286    TclEvalObjvInternal:cmd-entry   after
  0  63286    TclEvalObjvInternal:cmd-entry   exit
```

Tcl Scripts

The scripts included in Table 8-12 are from or based on scripts in the DTraceToolkit and have had comments trimmed to save space.

tcl_who.d

This script shows who is executing how much Tcl, in terms of Tcl commands.

Script

The `-Z` option is used so that this script can begin running before any instances of Tcl—and so before there are any tcl probes available to trace.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet
```

```
dtrace:::BEGIN
{
      printf("Tracing Tcl... Hit Ctrl-C to end.\n");
}

tcl*:::cmd-entry
{
      @calls[pid, uid, curpsinfo->pr_psargs] = count();
}

dtrace:::END
{
      printf("   %6s %6s %6s %-55s\n", "PID", "UID", "CMDS "ARGS");
      printa("   %6d %6d %@6d %-55.55s\n", @calls);
}
```

Examples

While tracing, the `func_abc.tcl` program was executed in another shell window:

```
# tcl_who.d
Tracing Tcl... Hit Ctrl-C to end.
^C
      PID    UID   CMDS ARGS
   123172    100     82 ./tclsh /opt/DTT/Code/Tcl/func_abc.tcl
```

This has traced the `func_abc.tcl` performing 82 commands. If multiple Tcl programs were running on the system, this script could identify the busiest, in terms of calls executed.

tcl_calls.d

This script counts Tcl procedure and command calls from any running Tcl program on the system, which has the tcl provider.

Script

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing Tcl... Hit Ctrl-C to end.\n");
}

tcl*:::proc-entry
{
      @calls[pid, "proc", copyinstr(arg0)] = count();
}

```

continues

```
tcl*:::cmd-entry
{
      @calls[pid, "cmd", copyinstr(arg0)] = count();
}

dtrace:::END
{
      printf(" %6s %-8s %-52s %8s\n", "PID", "TYPE", "NAME", "COUNT");
      printa(" %6d %-8s %-52s %@8d\n", @calls);
}
```

Examples

The `func_abc.tcl` program was traced using `tcl_calls.d`:

```
# tcl_calls.d
Tracing... Hit Ctrl-C to end.
^C
    PID TYPE     NAME                                                    COUNT
  16021 cmd      concat                                                      1
  16021 cmd      exit                                                        1
  16021 cmd      func_a                                                      1
  16021 cmd      func_b                                                      1
  16021 cmd      func_c                                                      1
[...]
  16021 proc     func_a                                                      1
  16021 proc     func_b                                                      1
  16021 proc     func_c                                                      1
  16021 proc     tclInit                                                     1
[...]
  16021 cmd      if                                                          8
  16021 cmd      info                                                       11
  16021 cmd      file                                                       12
  16021 cmd      proc                                                       12
```

The output has been truncated to fit; the procedure calls from the program can be seen as both executed commands (`TYPE "cmd"`) and procedures (`TYPE "proc"`), along with `tclInit()` to initialize the Tcl program. The commands that are not procedures are Tcl built-ins.

tcl_procflow.d

This program traces Tcl procedure flow, showing time stamps.

Script

Here's the script, with heading comment truncated to save space:

```
#!/usr/sbin/dtrace -Zs
[...]
#pragma D option quiet
#pragma D option switchrate=10

self int depth;
```

```

dtrace:::BEGIN
{
        printf("%3s %6s %-16s -- %s\n", "C", "PID", "TIME(us)", "PROCEDURE");
}

tcl*:::proc-entry
{
        printf("%3d %6d %-16d %*s-> %s\n", cpu, pid, timestamp / 1000,
            self->depth * 2, "", copyinstr(arg0));
        self->depth++;
}

tcl*:::proc-return
{
        self->depth -= self->depth > 0 ? 1 : 0;
        printf("%3d %6d %-16d %*s<- %s\n", cpu, pid, timestamp / 1000,
            self->depth * 2, "", copyinstr(arg0));
}
```

Example

The `func_abc.sh` program was traced using `tcl_procflow.d`:

```
# tcl_procflow.d
  C    PID TIME(us)         -- PROCEDURE
  0  16073 3904971507502    -> tclInit
  0  16073 3904971509096    <- tclInit
  0  16073 3904971509305    -> func_a
  0  16073 3904972511039      -> func_b
  0  16073 3904973521023        -> func_c
  0  16073 3904974530998        <- func_c
  0  16073 3904974531008      <- func_b
  0  16073 3904974531014    <- func_a
^C
```

As each procedure starts, the last column is indented by two spaces. This shows which procedure is calling which. The previous output begins with an `init` procedure and then shows that `func_a` began and then called `func_b`.

The columns are for CPU, PID, time since boot, indicator (`->` for procedure entry, and `<-` for procedure return), and procedure name.

If the output looks shuffled, check the CPU `C` and `TIME` columns, and postsort based on `TIME` if necessary.

By examining the `TIME(us)` column, latency in procedure flow can be identified as jumps in time. The script could be enhanced to show this as an extra column for the delta time between lines of output.

See Also: tcl_flowtime.d, tcl_syscolors.d

The `tcl_flowtime.d` script from the DTraceToolkit is similar to `tcl_procflow.d`, tracing both commands and procedures with a `DELTA(us)` column to help identify sources of latency. Another similar script from the DTraceToolkit is `tcl_syscolors.d`, which includes system calls in the output and uses terminal escape sequences to highlight different event types in different colors. They are similar to the Perl versions, `pl_flowtime.d` and `pl_syscolors.d`, which are demonstrated in the "Perl Scripts" section under `pl_flowinfo.d`.

See Also

The DTraceToolkit has other scripts for the tcl provider, including `tcl_calltime.d` for a report of inclusive and exclusive function time, and variants. See the Perl versions of these scripts in the "Perl Scripts" section for similar example output.

tcl_insflow.d

A script that is unique to the Tcl collection from the DTraceToolkit is `tcl_insflow.d`, which shows the flow of Tcl instructions for the processing of commands and procedures:

```
# tcl_insflow.d
  C    PID TIME(us)         DELTA(us)  TYPE -- CALL
  0 174829 4436207514685           3   cmd -> if
  0 174829 4436207514793         107  inst   -> push1
  0 174829 4436207514805          11  inst   <- push1
[...]
  0 174829 4436207522723           8   cmd -> func_a
  0 174829 4436207522742          18  proc   -> func_a
  0 174829 4436207522752          10  inst     -> push1
  0 174829 4436207522757           5  inst     <- push1
  0 174829 4436207522763           5  inst     -> push1
  0 174829 4436207522769           5  inst     <- push1
  0 174829 4436207522775           5  inst     -> invokeStk1
  0 174829 4436207522781           6   cmd       -> puts
  0 174829 4436207523212         430   cmd       <- puts
  0 174829 4436207523266          54  inst     <- invokeStk1
  0 174829 4436207523275           8  inst     -> pop
[...]
```

The output includes timing for latency analysis.

Summary

With the availability of language providers, software execution can be traced using DTrace, allowing the identification of frequently called or slow functions, object allocation, and errors and also as a way to study software flow. DTrace also allows events from across the software stack to be examined in the same context of the application, including disk and network I/O, CPU cross calls, and memory allocation. Chapter 9 continues the analysis of software, without the assumption that source code is available.

9

Applications

DTrace has the ability to follow the operation of applications from within the application source code, through system libraries, through system calls, and into the kernel. This visibility allows the root cause of issues (including performance issues) to be found and quantified, even if it is internal to a kernel device driver or something else outside the boundaries of the application code. Using DTrace, questions such as the following can be answered.

What transactions are occurring? With what latency?

What disk I/O is the application performing? What network I/O?

Why is the application on-CPU?

As an example, the following one-liner frequency counts application stack traces when the Apache Web server (`httpd`) performs the `read()` system call:

```
# dtrace -n 'syscall::read:entry /execname == "httpd"/ { @[ustack()] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
[...]

              libc.so.1`__read+0x7
              libapr-1.so.0.3.9`apr_socket_recv+0xb0
              libaprutil-1.so.0.3.9`socket_bucket_read+0x5b
              httpd`ap_core_input_filter+0x294
              mod_ssl.so`bio_filter_in_read+0xbc
              libcrypto.so.0.9.8`BIO_read+0xaf
              libssl.so.0.9.8`ssl3_get_record+0xb5
              libssl.so.0.9.8`ssl3_read_n+0x144
```

continues

```
              libssl.so.0.9.8`ssl3_read_bytes+0x161
              libssl.so.0.9.8`ssl3_read_internal+0x66
              libssl.so.0.9.8`ssl3_read+0x16
              libssl.so.0.9.8`SSL_read+0x42
              mod_ssl.so`ssl_io_input_read+0xf0
              mod_ssl.so`ssl_io_filter_input+0xd0
              httpd`ap_rgetline_core+0x66
              httpd`ap_read_request+0x1d1
              httpd`ap_process_http_connection+0xe4
              httpd`ap_run_process_connection+0x28
              httpd`child_main+0x3d8
              httpd`make_child+0x86
              httpd`ap_mpm_run+0x410
              httpd`main+0x812
              httpd`_start+0x7d
               31
```

The output has been truncated to show only the last stack trace. This stack trace was responsible for calling `read()` 31 times and shows the application code path through libssl (the Secure Sockets Layer library, because this was an HTTPS read). Each of the functions shown by the stack trace can be traced separately using DTrace, including function arguments, return value, and time.

The previous chapter focused on the programming languages of application software, particularly for developers who have access to the source code. This chapter focuses on application analysis for end users, regardless of language or layer in the software stack.

Capabilities

DTrace is capable of tracing every layer of the software stack, including examining the interactions of the various layers (see Figure 9-1).

Strategy

To get started using DTrace to examine applications, follow these steps (the target of each step is in bold):

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow and from the other chapters in the “See Also” section (which includes disk, file system, and network I/O).
2. In addition to those DTrace tools, familiarize yourself with any existing **application logs** and **statistics** that are available and also by any add-ons. (For example, before diving into Mozilla Firefox performance, try add-ons for performance analysis.) The information that these retrieve can show what is useful to investigate further with DTrace.

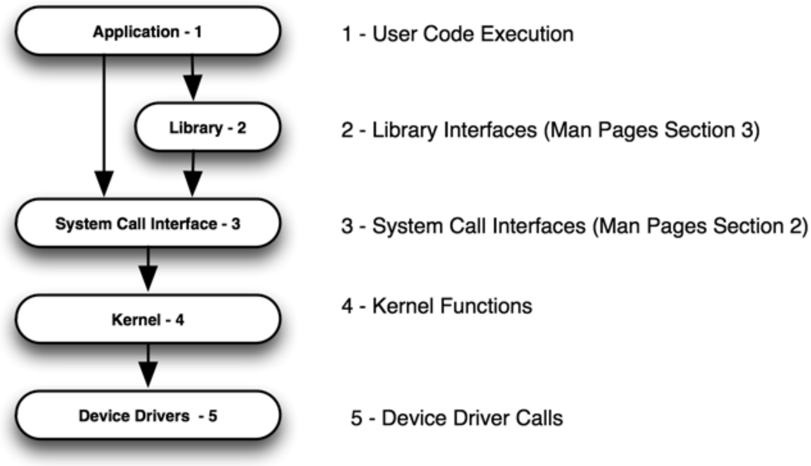


Figure 9-1 Software stack

3. Check whether any application **USDT providers** are available (for example, the mozilla provider for Mozilla Firefox).
4. Examine application behavior using the **syscall** provider, especially if the application has a high system CPU time. This is often an effective way to get a high-level picture of what the application is doing by examining what it is requesting the kernel to do. System call entry arguments and return errors can be examined for troubleshooting issues, and system call latency can be examined for performance analysis.
5. Examine application behavior in the context of **system resources**, such as CPUs, disks, file systems, and network interfaces. Refer to the appropriate chapter in this book.
6. Write tools to generate **known workloads**, such as performing a client transaction. It can be extremely helpful to have a known workload to refer to while developing DTrace scripts.
7. Familiarize yourself with application internals. Sources may include application documentation and source code, if available. DTrace can also be used to learn the internals of an application, such as by examining **stack traces** whenever the application performs I/O (see the example at the start of this chapter).
8. Use a **language provider** to trace application code execution, if one exists and is available (for example, perl). See Chapter 8, Languages.

9. Use the **pid provider** to trace the internals of the application software and libraries it uses, referring to the source code if available. Write scripts to examine higher-level details first (operation counts), and drill down deeper into areas of interest.

Checklist

Consider Table 9-1 a checklist of application issue types that can be examined using DTrace. This is similar to the checklist in Chapter 8 but is in terms of applications rather than the language.

Table 9-1 Applications Checklist

Issue	Description
on-CPU time	An application is hot on-CPU, showing high %CPU in `top(1)` or `prstat(1M)`. DTrace can identify the reason by sampling user stack traces with the profile provider and by tracing application functions with vtime-stamps. Reasons for high on-CPU time may include the following: • Compression • Encryption • Dataset iteration (code path loops) • Spin lock contention • Memory I/O The actual make-up of CPU time, whether it is cycles on core (for example, for the Arithmetic Logic Unit) or cycles while stalled (for example, waiting for memory bus I/O) can be investigated further using the DTrace cpc provider, if available.
off-CPU time	Applications will spend time off-CPU while waiting for I/O, waiting for locks (not spinning), and while waiting to be dispatched on a CPU after returning to the ready to run state. These events can be examined and timed with DTrace, such as by using the sched provider to look at thread events. Time off-CPU during I/O, especially disk or network I/O, is a common cause of performance issues (for example, an application performing file system reads served by slow disks, or a DNS lookup during client login, waiting on network I/O to the DNS server). When interpreting off-CPU time, it is important to differentiate between time spent off-CPU because of the following: • Waiting on I/O during an application transaction • Waiting for work to do Applications may spend most of their time waiting for work to do, which is not typically a problem.

Table 9-1 Applications Checklist (*Continued*)

Issue	Description
Volume	Applications may be calling a particular function or code path too frequently; this is the simplest type of issue to DTrace: frequency count function calls. Examining function arguments may identify other inefficiencies, such as performing I/O with small byte sizes when larger sizes should be possible.
Locks	Waiting on locks can occur both on-CPU (spin) and off-CPU (wait). Locks are used for synchronization of multithreaded applications and, when poorly used, can cause application latency and thread serialization. Use DTrace to examine lock usage using the plockstat provider if available or using pid or profile.
Memory Allocation	Memory allocation can be examined in situations when applications consume excessive amounts of memory. Calls to manage memory (such as `malloc()`) can be traced, along with entry and return arguments.
Errors	Applications can encounter errors in their own code and from system libraries and system calls that they execute. Encountering errors is normal for software, which should be written to handle them correctly. However, it is possible that errors are being encountered but not handled correctly by the application. DTrace can be used to examine whether errors are occurring and, if so, their origin.

Providers

Table 9-2 shows providers of most interest when tracing applications.

Table 9-2 Providers for Applications

Provider	Description
proc	Trace application process and thread creation and destruction and signals.
syscall	Trace entry and return of operating system calls, arguments, and return values.
profile	Sample application CPU activity at a custom rate.
sched	Trace application thread scheduling events.
vminfo	Virtual memory statistic probes, based on `vmstat(1M)` statistics.
sysinfo	Kernel statistics probes, based on `mpstat(1M)` statistics.
plockstat	Trace user-land lock events.
cpc	CPU Performance Counters provider, for CPU cache hit/miss by function.
pid	Trace internals of the application including calls to system libraries.
language	Specific language provider: See Chapter 8.

You can find complete lists of provider probes and arguments in the DTrace Guide.[1]

pid Provider

The Process ID (pid) provider instruments user-land function execution, providing probes for function entry and return points and for every instruction in the function. It also provides access to function arguments, return codes, return instruction offsets, and register values. By tracing function entry and return, the elapsed time and on-CPU time during function execution can also be measured. It is available on Solaris and Mac OS X and is currently being developed for FreeBSD.[2]

The pid provider is associated with a particular process ID, which is part of the provider name: pid<PID>. The PID can be written literally, such as pid123, or specified using the macro variable `$target`, which provides the PID when either the `-p PID` or `-c command` option is used.

Listing pid provider function entry probes for the bash shell (running as PID 1122) yields the following:

```
# dtrace -ln 'pid$target:::entry' -p 1122
   ID   PROVIDER            MODULE                          FUNCTION NAME
12539    pid1122              bash                            _start entry
12540    pid1122              bash                             __fsr entry
12541    pid1122              bash                              main entry
12542    pid1122              bash                parse_long_options entry
12543    pid1122              bash               parse_shell_options entry
12544    pid1122              bash                        exit_shell entry
12545    pid1122              bash                           sh_exit entry
12546    pid1122              bash                  execute_env_file entry
12547    pid1122              bash                 run_startup_files entry
12548    pid1122              bash               shell_is_restricted entry
12549    pid1122              bash             maybe_make_restricted entry
12550    pid1122              bash                            uidget entry
12551    pid1122              bash                 disable_priv_mode entry
12552    pid1122              bash                       run_wordexp entry
12553    pid1122              bash                   run_one_command entry
[...]
15144    pid1122    libcurses.so.1                            addstr entry
15145    pid1122    libcurses.so.1                           attroff entry
15146    pid1122    libcurses.so.1                            attron entry
15147    pid1122    libcurses.so.1                           attrset entry
15148    pid1122    libcurses.so.1                              beep entry
15149    pid1122    libcurses.so.1                              bkgd entry
[...]
15704    pid1122    libsocket.so.1                         endnetent entry
15705    pid1122    libsocket.so.1                       getnetent_r entry
15706    pid1122    libsocket.so.1                        str2netent entry
15707    pid1122    libsocket.so.1                    getprotobyname entry
```

1. This is currently at *http://wikis.sun.com/display/DTrace/Documentation*.
2. This is by Rui Paulo for the DTrace user-land project: *http://freebsdfoundation.blogspot.com/2010/06/dtrace-userland-project.html*.

```
15708    pid1122    libsocket.so.1                    getprotobynumber entry
15709    pid1122    libsocket.so.1                         getprotoent entry
[...]
19019    pid1122         libc.so.1                               fopen entry
19020    pid1122         libc.so.1                       _freopen_null entry
19021    pid1122         libc.so.1                             freopen entry
19022    pid1122         libc.so.1                             fgetpos entry
19023    pid1122         libc.so.1                             fsetpos entry
19024    pid1122         libc.so.1                               fputc entry
[...]
```

There were 8,003 entry probes listed. The previous truncated output shows a sample of the available probes from the bash code segment and three libraries: libcurses, libsocket, and libc. The probe module name is the segment name.

Listing all pid provider probes for the libc function `fputc()` yields the following:

```
# dtrace -ln 'pid$target::fputc:' -p 1122
   ID   PROVIDER            MODULE                          FUNCTION NAME
19024    pid1122         libc.so.1                             fputc entry
20542    pid1122         libc.so.1                             fputc return
20543    pid1122         libc.so.1                             fputc 0
20544    pid1122         libc.so.1                             fputc 1
20545    pid1122         libc.so.1                             fputc 3
20546    pid1122         libc.so.1                             fputc 4
20547    pid1122         libc.so.1                             fputc 7
20548    pid1122         libc.so.1                             fputc c
20549    pid1122         libc.so.1                             fputc d
20550    pid1122         libc.so.1                             fputc 13
20551    pid1122         libc.so.1                             fputc 16
20552    pid1122         libc.so.1                             fputc 19
20553    pid1122         libc.so.1                             fputc 1c
20554    pid1122         libc.so.1                             fputc 21
20555    pid1122         libc.so.1                             fputc 24
20556    pid1122         libc.so.1                             fputc 25
20557    pid1122         libc.so.1                             fputc 26
```

The probes listed are the entry and return probes for the `fputc()` function, as well as probes for each instruction offset in hexadecimal (0, 1, 3, 4, 7, c, d, and so on).

Be careful when using the pid provider, especially in production environments. Application processes vary greatly in size, and many production applications have large text segments with a large number of instrumentable functions, each with tens to hundreds of instructions and with each instruction another potential probe target for the pid provider. The invocation `dtrace -n 'pid1234:::'` will instruct DTrace to instrument every function entry and return and to instrument every instruction in process PID 1234. Here's an example:

```
solaris# dtrace -n 'pid1471:::'
dtrace: invalid probe specifier pid1471:::: failed to create offset probes in
'__1cFStateM_sub_Op_ConI6MpknENode__v_': Not enough space

solaris# dtrace -n 'pid1471:::entry'
dtrace: description 'pid1471:::entry' matched 26847 probes
```

Process PID 1471 was a Java JVM process. The first DTrace command attempted to insert a probe at every instruction location in the JVM but was unable to complete. The `Not enough space` error means the default number of 250,000 pid provider probes was not enough to complete the instrumentation. The second invocation in the example instruments the same process, but this time with the entry string in the name component of the probe, instructing DTrace to insert a probe at the entry point of every function in the process. In this case, DTrace found 26,847 instrumentation points.

Once a process is instrumented with the pid provider, depending on the number of probes and how busy the process is, using the pid provider will induce some probe effect, meaning it can slow the execution speed of the target process, in some cases dramatically.

Stability

The pid provider is considered an *unstable* interface, meaning that the provider interface (which consists of the probe names and arguments) may be subject to change between application software versions. This is because the interface is dynamically constructed based on the thousands of compiled functions that make up a software application. It is these functions that are subject to change, and when they do, so does the pid provider. This means that any DTrace scripts or one-liners based on the pid provider may be dependent on the application software version they were written for.

Although application software can and is likely to change between versions, many library interfaces are likely to remain unchanged, such as libc, libsocket, libpthread, and many others, especially those exporting standard interfaces such as POSIX. These can make good targets for tracing with the pid provider, because one-liners and scripts will have a higher degree of stability than when tracing application-specific software.

If a pid-based script has stopped working because of minor software changes, then ideally the script can be repaired with equivalent minor changes to match the newer software. If the software has changed significantly, then the pid-based script may need to be rewritten entirely. Because of this instability, it is recommended to use pid only when needed. If there are stable providers available that can serve a similar role, they should be used instead, and the scripts that use them will not need to be rewritten as the software changes.

Since pid is an unstable interface, the pid provider one-liners and scripts in this book are not guaranteed to work or be supported by software vendors.

The pid provider scripts in this book serve not just as examples of using the pid provider in D programs but also as example data that DTrace can make available and why that can be useful. If these scripts stop working, you can try fixing them or check for updated versions on the Web (try this book's Web site, *www.dtracebook.com*).

Arguments and Return Value

The arguments and return value for functions can be inspected on the pid entry and return probes.

`pid<PID>:::entry`: The function arguments is (`uint64_t`) `arg0 ... argn`.

`pid<PID>:::return`: The program counter is (`uint64_t`) `arg0`; the return value is (uint64_t) `arg1`.

The `uregs[]` array can also be accessed to examine individual user registers.

cpc Provider

The CPU Performance Counter (cpc) provider provides probes for profiling CPU events, such as instructions, cache misses, and stall cycles. These CPU events are based on the performance counters that the CPUs provide, which vary between manufacturers, types, and sometimes versions of the same type of CPU. A generic interface for the performance counters has been developed, the Performance Application Programming Interface (PAPI),[3] which is supported by the cpc provider in addition to the platform-specific counters. The cpc provider is fully documented in the cpc provider section of the DTrace Guide and is currently available only in Solaris Nevada.[4]

The cpc provider probe names have the following format:

```
cpc:::<event name>-<mode>-<optional mask-><count>
```

The event name may be a PAPI name or a platform-specific event name. On Solaris, events for the current CPU type can be listed using `cpustat(1M)`:

```
solaris# cpustat -h
Usage:
        cpustat [-c events] [-p period] [-nstD] [-T d|u] [interval [count]]
[...]
        Generic Events:
```
continues

3. See *http://icl.cs.utk.edu/papi*.

4. This was integrated in snv_109, defined by PSARC 2008/480, and developed by Jon Haslam. See his blog post about cpc, currently at *http://blogs.sun.com/jonh/entry/finally_dtrace_meets_the_cpu*.

```
        event[0-3]: PAPI_br_ins PAPI_br_msp PAPI_br_tkn PAPI_fp_ops
                 PAPI_fad_ins PAPI_fml_ins PAPI_fpu_idl PAPI_tot_cyc
                 PAPI_tot_ins PAPI_l1_dca PAPI_l1_dcm PAPI_l1_ldm
                 PAPI_l1_stm PAPI_l1_ica PAPI_l1_icm PAPI_l1_icr
                 PAPI_l2_dch PAPI_l2_dcm PAPI_l2_dcr PAPI_l2_dcw
                 PAPI_l2_ich PAPI_l2_icm PAPI_l2_ldm PAPI_l2_stm
                 PAPI_res_stl PAPI_stl_icy PAPI_hw_int PAPI_tlb_dm
                 PAPI_tlb_im PAPI_l3_dcr PAPI_l3_icr PAPI_l3_tcr
                 PAPI_l3_stm PAPI_l3_ldm PAPI_l3_tcm

        See generic_events(3CPC) for descriptions of these events

        Platform Specific Events:

        event[0-3]: FP_dispatched_fpu_ops FP_cycles_no_fpu_ops_retired
                 FP_dispatched_fpu_ops_ff LS_seg_reg_load
                 LS_uarch_resync_self_modify LS_uarch_resync_snoop
                 LS_buffer_2_full LS_locked_operation LS_retired_cflush
                 LS_retired_cpuid DC_access DC_miss DC_refill_from_L2
                 DC_refill_from_system DC_copyback DC_dtlb_L1_miss_L2_hit
                 DC_dtlb_L1_miss_L2_miss DC_misaligned_data_ref
[...]
        See "BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h
        Processors" (AMD publication 31116)
```

The first group, `Generic Events`, is the PAPI events and is documented on Solaris in the generic_events(3CPC) man page. The second group, `Platform Specific Events`, is from the CPU manufacturer and is typically documented in the CPU user guide referenced in the `cpustat(1M)` output.

The mode component of the probe name can be `user` for profiling user-mode, `kernel` for kernel-mode, or `all` for both.

The optional mask component is sometimes used by platform-specific events, as directed by the CPU user guide.

The final component of the probe name is the overflow count: Once this many of the specified event has occurred on the CPU, the probe fires on that CPU. For frequent events, such as cycle and instruction counts, this can be set to a high number to reduce the rate that the probe fires and therefore reduce the impact on target application performance.

cpc provider probes have two arguments: `arg0` is the kernel program counter or 0 if not executing in the kernel, and `arg1` is the user-level program counter or 0 if not executing in user-mode.

Depending on the CPU type, it may not be possible to enable more than one cpc probe simultaneously. Subsequent enablings will encounter a `Failed to enable probe` error. This behavior is similar to, and for the same reason as, the operating system, allowing only one invocation of `cpustat(1M)` at a time. There is a finite number of performance counter registers available for each CPU type.

The sections that follow have example cpc provider one-liners and output.

See Also

There are many topics relevant to application analysis, most of which are covered fully in separate chapters of this book.

Chapter 3: System View
Chapter 4: Disk I/O
Chapter 5: File Systems
Chapter 6: Network Lower-Level Protocols
Chapter 7: Application-Level Protocols
Chapter 8: Languages

All of these can be considered part of this chapter. The one-liners and scripts that follow summarize application analysis with DTrace and introduce some remaining topics such as signals, thread scaling, and the cpc provider.

One-Liners

For many of these, a Web server with processes named `httpd` is used as the target application. Modify `httpd` to be the name of the application process of interest.

proc provider

Trace new processes:

```
dtrace -n 'proc:::exec-success { trace(execname); }'
```

Trace new processes (current FreeBSD[5]):

```
dtrace -n 'proc:::exec_success { trace(execname); }'
```

New processes (with arguments):

```
dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
```

5. FreeBSD 8.0; this will change to become `exec-success` (consistent with Solaris and Mac OS X), now that support for hyphens in FreeBSD probe names is being developed.

New threads created, by process:

```
dtrace -n 'proc:::lwp-create { @[pid, execname] = count(); }'
```

Successful signal details:

```
dtrace -n 'proc:::signal-send { printf("%s -%d %d", execname, args[2], args[1]->pr_pid); }'
```

syscall provider

System call counts for processes named `httpd`:

```
dtrace -n 'syscall:::entry /execname == "httpd"/ { @[probefunc] = count(); }'
```

System calls with non-zero `errno` (errors):

```
dtrace -n 'syscall:::return /errno/ { @[probefunc, errno] = count(); }'
```

profile provider

User stack trace profile at 101 Hertz, showing process name and stack:

```
dtrace -n 'profile-101 { @[execname, ustack()] = count(); }'
```

User stack trace profile at 101 Hertz, showing process name and top five stack frames:

```
dtrace -n 'profile-101 { @[execname, ustack(5)] = count(); }'
```

User stack trace profile at 101 Hertz, showing process name and stack, top ten only:

```
dtrace -n 'profile-101 { @[execname, ustack()] = count(); } END { trunc(@, 10); }'
```

User stack trace profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[ustack()] = count(); }'
```

User function name profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[ufunc(arg1)] = count(); }'
```

User module name profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[umod(arg1)] = count(); }'
```

sched provider

Count user stack traces when processes named `httpd` leave CPU:

```
dtrace -n 'sched:::off-cpu /execname == "httpd"/ { @[ustack()] = count(); }'
```

pid provider

The pid provider instruments functions from a particular software version; these example one-liners may therefore require modifications to match the software version you are running. They can be executed on an existing process by using `-p PID` or by running a new process using `-c command`.

Count process segment function calls:

```
dtrace -n 'pid$target:a.out::entry { @[probefunc] = count(); }' -p PID
```

Count libc function calls:

```
dtrace -n 'pid$target:libc::entry { @[probefunc] = count(); }' -p PID
```

Count libc string function calls:

```
dtrace -n 'pid$target:libc:str*:entry { @[probefunc] = count(); }' -p PID
```

Trace libc `fsync()` calls showing file descriptor:

```
dtrace -n 'pid$target:libc:fsync:entry { trace(arg0); }' -p PID
```

Trace libc `fsync()` calls showing file path name:

```
dtrace -n 'pid$target:libc:fsync:entry { trace(fds[arg0].fi_pathname); }' -p PID
```

Count requested `malloc()` bytes by user stack trace:

```
dtrace -n 'pid$target::malloc:entry { @[ustack()] = sum(arg0); }' -p PID
```

Trace failed `malloc()` requests:

```
dtrace -n 'pid$target::malloc:return /arg1 == NULL/ { ustack(); }' -p PID
```

See the "C" section of Chapter 8 for more pid provider one-liners.

plockstat provider

As with the pid provider, these can also be run using the `-c` command.

Mutex blocks by user-level stack trace:

```
dtrace -n 'plockstat$target:::mutex-block { @[ustack()] = count(); }' -p PID
```

Mutex spin counts by user-level stack trace:

```
dtrace -n 'plockstat$target:::mutex-acquire /arg2/ { @[ustack()] = sum(arg2); }' -p PID
```

Reader/writer blocks by user-level stack trace:

```
dtrace -n 'plockstat$target:::rw-block { @[ustack()] = count(); }' -p PID
```

cpc provider

These cpc provider one-liners are dependent on the availability of both the cpc provider and the event probes (for Solaris, see `cpustat(1M)` to learn what events are available on your system). The following overflow counts (200,000; 50,000; and 10,000) have been picked to balance between the rate of events and fired DTrace probes.

User-mode instructions by process name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname] = count(); }'
```

User-mode instructions by process name and function name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode instructions for processes named `httpd` by function name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 /execname == "httpd"/ { @[ufunc(arg1)] =
count(); }'
```

User-mode CPU cycles by process name and function name:

```
dtrace -n 'cpc:::PAPI_tot_cyc-user-200000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one instruction cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_icm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one data cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_dcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-two cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-three cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l3_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode conditional branch misprediction by process name and function name:

```
dtrace -n 'cpc:::PAPI_br_msp-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode resource stall cycles by process name and function name:

```
dtrace -n 'cpc:::PAPI_res_stl-user-50000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode floating-point operations by process name and function name:

```
dtrace -n 'cpc:::PAPI_fp_ops-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode TLB misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_tlb_tl-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

One-Liner Selected Examples

There are additional examples of one-liners in the "Case Study" section.

New Processes (with Arguments)

New processes were traced on Solaris while the `man ls` command was executed:

```
solaris# dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
dtrace: description 'proc:::exec-success ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  13487         exec_common:exec-success   man ls
  0  13487         exec_common:exec-success   sh -c cd /usr/share/man; tbl /usr/share/
man/man1/ls.1 |neqn /usr/share/lib/pub/
```

```
  0  13487          exec_common:exec-success   tbl /usr/share/man/man1/ls.1
  0  13487          exec_common:exec-success   neqn /usr/share/lib/pub/eqnchar -
  0  13487          exec_common:exec-success   nroff -u0 -Tlp -man -
  0  13487          exec_common:exec-success   col -x
  0  13487          exec_common:exec-success  sh -c trap '' 1 15; /usr/bin/mv -f /tmp/
mpcJaP5g /usr/share/man/cat1/ls.1 2> /d
  0  13487          exec_common:exec-success  /usr/bin/mv -f /tmp/mpcJaP5g /usr/share/
man/cat1/ls.1
  0  13487          exec_common:exec-success   sh -c more -s /tmp/mpcJaP5g
  0  13487          exec_common:exec-success   more -s /tmp/mpcJaP5g
^C
```

The variety of programs that are executed to process `man ls` are visible, ending with the `more(1)` command that shows the man page.

Mac OS X currently doesn't provide the full argument list in `pr_psargs`, which is noted in the comments of the curpsinfo translator:

```
macosx# grep pr_psargs /usr/lib/dtrace/darwin.d
      char pr_psargs[80];     /* initial characters of arg list */
      pr_psargs = P->p_comm; /* XXX omits command line arguments XXX */
      pr_psargs = xlate <psinfo_t> ((struct proc *)(T->task->bsd_info)).pr_psargs; /*

XXX omits command line arguments XXX */
```

And using `pr_psargs` in `trace()` on Mac OS X can trigger `tracemem()` behavior, printing hex dumps from the address, which makes reading the output a little difficult. It may be easier to just use the `execname` for this one-liner for now. Here's an example of tracing `man ls` on Mac OS X:

```
macosx# dtrace -n 'proc:::exec-success { trace(execname); }'
dtrace: description 'proc:::exec-success ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  19374          posix_spawn:exec-success   sh
  0  19374          posix_spawn:exec-success   sh
  0  19368         __mac_execve:exec-success   sh
  0  19368         __mac_execve:exec-success   tbl
  0  19368         __mac_execve:exec-success   sh
  0  19368         __mac_execve:exec-success   grotty
  0  19368         __mac_execve:exec-success   more
  1  19368         __mac_execve:exec-success   man
  1  19368         __mac_execve:exec-success   sh
  1  19368         __mac_execve:exec-success   gzip
  1  19368         __mac_execve:exec-success   gzip
  1  19374          posix_spawn:exec-success   sh
  1  19368         __mac_execve:exec-success   groff
  1  19368         __mac_execve:exec-success   troff
  1  19368         __mac_execve:exec-success   gzip
^C
```

Note that the output is shuffled (the CPU ID change is a hint). For the correct order, include a time stamp in the output and postsort.

System Call Counts for Processes Called httpd

The Apache Web server runs multiple `httpd` processes to serve Web traffic. This can be a problem for traditional system call debuggers (such as `truss(1)`), which can examine only one process at a time, usually by providing a process ID. DTrace can examine all processes simultaneously, making it especially useful for multiprocess applications such as Apache.

This one-liner frequency counts system calls from all running Apache `httpd` processes:

```
solaris# dtrace -n 'syscall:::entry /execname == "httpd"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 225 probes
^C

  accept                                                          1
  getpid                                                          1
  lwp_mutex_timedlock                                             1
  lwp_mutex_unlock                                                1
  shutdown                                                        1
  brk                                                             4
  gtime                                                           5
  portfs                                                          7
  mmap64                                                         10
  waitsys                                                        30
  munmap                                                         33
  doorfs                                                         39
  openat                                                         49
  writev                                                         51
  stat64                                                         60
  close                                                          61
  fcntl                                                          73
  read                                                           74
  lwp_sigmask                                                    78
  getdents64                                                     98
  pollsys                                                       100
  fstat64                                                       109
  open64                                                        207
  lstat64                                                       245
```

The most frequently called system call was `lstat64()`, called 245 times.

User Stack Trace Profile at 101 Hertz, Showing Process Name and Top Five Stack Frames

This one-liner is a quick way to see not just who is on-CPU but what they are doing:

```
solaris# dtrace -n 'profile-101 { @[execname, ustack(5)] = count(); }'
dtrace: description 'profile-101 ' matched 1 probe
^C
[...]
  mpstat
              libc.so.1`p_online+0x7
```

```
              mpstat`acquire_snapshot+0x131
              mpstat`main+0x27d
              mpstat`_start+0x7d
               13
  httpd
              libc.so.1`__forkx+0xb
              libc.so.1`fork+0x1d
              mod_php5.2.so`zif_proc_open+0x970
              mod_php5.2.so`execute_internal+0x45
              mod_php5.2.so`dtrace_execute_internal+0x59
               42
  sched
              541
```

No stack trace was shown for sched (the kernel), since this one-liner is examining user-mode stacks (`ustack()`), not kernel stacks (`stack()`). This could be eliminated from the output by adding the predicate `/arg1/` (check that the user-mode program counter is nonzero) to ensure that only user stacks are sampled.

User-Mode Instructions by Process Name

To introduce this one-liner, a couple of test applications were written and executed called app1 and app2, each single-threaded and running a continuous loop of code. Examining these applications using `top(1)` shows the following:

```
last pid:  4378;  load avg:  2.13,  2.00,  1.62;  up 4+02:53:19       06:24:05
98 processes: 95 sleeping, 3 on cpu
CPU states: 73.9% idle, 25.2% user,  0.9% kernel,  0.0% iowait,  0.0% swap
Kernel: 866 ctxsw, 19 trap, 1884 intr, 2671 syscall
Memory: 32G phys mem, 1298M free mem, 4096M total swap, 4096M free swap

   PID USERNAME NLWP PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
  4319 root        1  10    0 1026M  513M cpu/3   10:50 12.50% app2
  4318 root        1  10    0 1580K  808K cpu/7   10:56 12.50% app1
[...]
```

`top(1)` reports that each application is using 12.5 percent of the total CPU capacity, which is a single core on this eight-core system. The Solaris `prstat -mL` breaks down the CPU time into microstates and shows this in terms of a single thread:

```
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  4318 root     100 0.0 0.0 0.0 0.0 0.0 0.0 0.0   0   8   0   0 app1/1
  4319 root     100 0.0 0.0 0.0 0.0 0.0 0.0 0.0   0   8   0   0 app2/1
[...]
```

`prstat(1M)` shows that each thread is running at 100 percent user time (USR). This is a little more information than simply %CPU from `top(1)`, and it indicates that these applications are both spending time executing their own code.

The cpc provider allows %CPU time to be understood in greater depth. This one-liner uses the cpc provider to profile instructions by process name. The probe specified fires for every 200,000th user-level instruction, counting the current process name at the time:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname] = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-user-200000 ' matched 1 probe
^C

  sendmail                                                   1
  dtrace                                                     2
  mysqld                                                     6
  sshd                                                       7
  nscd                                                      14
  httpd                                                     16
  prstat                                                    23
  mpstat                                                    52
  app2                                                     498
  app1                                                  154801
```

So, although the output from `top(1)` and `prstat(1M)` suggests that both applications are very similar in terms of CPU usage, the cpc provider shows that they are in fact very different. During the same interval, app1 executed roughly 300 times more CPU instructions than app2.

The other cpc one-liners can explain this further; app1 was written to continually execute fast register-based instructions, while app2 continually performs much slower main memory I/O.

User-Mode Instructions for Processes Named httpd by Function Name

This one-liner matches processes named `httpd` and profiles instructions by function, counting on every 200,000th instruction:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-user-200000 /execname == "httpd"/ {
@[ufunc(arg1)]  = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-user-200000 ' matched 1 probe
^C

  httpd`ap_invoke_handler                                    1
  httpd`pcre_exec                                            1
  libcrypto.so.0.9.8`SHA1_Update                             1
[...]
  libcrypto.so.0.9.8`bn_sqr_comba8                          39
  libz.so.1`crc32_little                                    41
  libcrypto.so.0.9.8`sha1_block_data_order                  50
  libcrypto.so.0.9.8`_x86_AES_encrypt                       88
  libz.so.1`compress_block                                 103
  libcrypto.so.0.9.8`bn_mul_add_words                      117
  libcrypto.so.0.9.8`bn_mul_add_words                      127
  libcrypto.so.0.9.8`bn_mul_add_words                      133
  libcrypto.so.0.9.8`bn_mul_add_words                      134
```

```
  libz.so.1`fill_window                                                    222
  libz.so.1`deflate_slow                                                   374
  libz.so.1`longest_match                                                 1022
```

The functions executing the most instructions are in the libz library, which performs compression.

User-Mode Level-Two Cache Misses by Process Name and Function Name

This example is included to suggest what to do when encountering this error:

```
solaris# dtrace -n 'cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
dtrace: invalid probe specifier cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] =
 count(); }: probe description cpc:::PAPI_l2_tcm-user-10000 does not match any probes
```

This system does have the cpc provider; however, this probe is invalid. After checking for typos, check whether the event name is supported on this system using `cpustat(1M)` (Solaris):

```
solaris# cpustat -h
Usage:
        cpustat [-c events] [-p period] [-nstD] [-T d|u] [interval [count]]
[...]
        Generic Events:

        event[0-3]: PAPI_br_ins PAPI_br_msp PAPI_br_tkn PAPI_fp_ops
                 PAPI_fad_ins PAPI_fml_ins PAPI_fpu_idl PAPI_tot_cyc
                 PAPI_tot_ins PAPI_l1_dca PAPI_l1_dcm PAPI_l1_ldm
                 PAPI_l1_stm PAPI_l1_ica PAPI_l1_icm PAPI_l1_icr
                 PAPI_l2_dch PAPI_l2_dcm PAPI_l2_dcr PAPI_l2_dcw
                 PAPI_l2_ich PAPI_l2_icm PAPI_l2_ldm PAPI_l2_stm
                 PAPI_res_stl PAPI_stl_icy PAPI_hw_int PAPI_tlb_dm
                 PAPI_tlb_im PAPI_l3_dcr PAPI_l3_icr PAPI_l3_tcr
                 PAPI_l3_stm PAPI_l3_ldm PAPI_l3_tcm

        See generic_events(3CPC) for descriptions of these events

        Platform Specific Events:

        event[0-3]: FP_dispatched_fpu_ops FP_cycles_no_fpu_ops_retired
[...]
```

This output shows that the PAPI_l2_tcm event (level-two cache miss) is not supported on this system. However, it also shows that PAPI_l2_dcm (level-two data cache miss) and PAPI_l2_icm (level-two instruction cache miss) are supported. Adjusting the one-liner for, say, data cache misses only is demonstrated by the following one-liner:

```
solaris# dtrace -n 'cpc:::PAPI_l2_dcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
dtrace: description 'cpc:::PAPI_l2_dcm-user-10000 ' matched 1 probe
^C

  dtrace                    libproc.so.1`byaddr_cmp                                1
  dtrace                    libproc.so.1`symtab_getsym                             1
  dtrace                    libc.so.1`memset                                       1
  mysqld                    mysqld`srv_lock_timeout_and_monitor_thread             1
  mysqld                    mysqld`sync_array_print_long_waits                     1
  dtrace                    libproc.so.1`byaddr_cmp_common                         2
  dtrace                    libc.so.1`qsort                                        2
  dtrace                    libproc.so.1`optimize_symtab                           3
  dtrace                    libproc.so.1`byname_cmp                                6
  dtrace                    libc.so.1`strcmp                                      17
  app2                      app2`main                                            399
```

This one-liner can then be run for instruction cache misses so that both types of misses can be considered.

Should the generic PAPI events be unavailable or unsuitable, the platform-specific events (as listed by `cpustat(1M)`) may allow the event to be examined, albeit in a way that is tied to the current CPU version.

Scripts

Table 9-3 summarizes the scripts that follow and the providers they use.

procsnoop.d

This is a script version of the "New Processes" one-liner shown earlier. Tracing the execution of new processes provides important visibility for applications that call

Table 9-3 Application Script Summary

Script	Description	Provider
`procsnoop`	Snoop process execution	proc
`procsystime`	System call time statistics by process	syscall
`uoncpu.d`	Profile application on-CPU user stacks	profile
`uoffcpu.d`	Count application off-CPU user stacks by time	sched
`plockstat`	User-level mutex and read/write lock statistics	plockstat
`kill.d`	Snoop process signals	syscall
`sigdist.d`	Signal distribution by source and destination processes	syscall
`threaded.d`	Sample multithreaded CPU usage	profile

the command line; some applications can call shell commands so frequently that it becomes a performance issue—one that is difficult to spot in traditional tools (such as `prstat(1M)` and `top(1)`) because the processes are so short-lived.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-8s %5s %6s %6s %s\n", "TIME(ms)", "UID", "PID", "PPID",
            "COMMAND");
        start = timestamp;
}

proc:::exec-success
{
        printf("%-8d %5d %6d %6d %s\n", (timestamp - start) / 1000000,
            uid, pid, ppid, curpsinfo->pr_psargs);
}
```

Script procsnoop.d

Example

The following shows the Oracle Solaris commands executed as a consequence of restarting the `cron` daemon via `svcadm(1M)`:

```
solaris# procsnoop.d
TIME(ms)   UID    PID   PPID COMMAND
3227         0  13273  12224 svcadm restart cron
3709         0  13274    106 /sbin/sh -c exec /lib/svc/method/svc-cron
3763         0  13274    106 /sbin/sh /lib/svc/method/svc-cron
3773         0  13275  13274 /usr/bin/rm -f /var/run/cron_fifo
3782         0  13276  13274 /usr/sbin/cron
```

The `TIME(ms)` column is printed so that the output can be postsorted if desired (DTrace may shuffle the output slightly because it collects buffers from multiple CPUs).

See Also: execsnoop

A program called `execsnoop` exists from the DTraceToolkit, which has similar functionality to that of `procsnoop`. It was written originally for Oracle Solaris and is now shipped on Mac OS X by default. `execsnoop` wraps the D script in the shell so that command-line options are available:

```
macosx# execsnoop -h
USAGE: execsnoop [-a|-A|-ehjsvZ] [-c command]
       execsnoop                # default output
                -a              # print all data
                -A              # dump all data, space delimited
                -e              # safe output, parseable
                -j              # print project ID
                -s              # print start time, us
                -v              # print start time, string
                -Z              # print zonename
                -c command      # command name to snoop
  eg,
        execsnoop -v            # human readable timestamps
        execsnoop -Z          # print zonename
        execsnoop -c ls         # snoop ls commands only
```

`execsnoop` traces process execution by tracing the `exec()` system call (and variants), which do differ slightly between operating systems. Unfortunately, system calls are not a stable interface, even across different versions of the same operating system. Small changes to `execsnoop` have been necessary to keep it working across different versions of Oracle Solaris, because of subtle changes with the names of the `exec()` system calls. The lesson here is to always prefer the stable providers, such as the proc provider (which is stable) instead of syscall (which isn't).

procsystime

`procsystime` is a generic system call time reporter. It can count the execution of system calls, their elapsed time, and on-CPU time and can produce a report showing the system call type and process details. It is from the DTraceToolkit and shipped on Mac OS X by default in `/usr/bin`.

Script

The essence of the script is explained here; the actual script is too long and too uninteresting (mostly dealing with command-line options) to list; see the DTraceToolkit for the full listing.

```
syscall:::entry
/self->ok/
{
      @Counts[probefunc] = count();
      self->start = timestamp;
      self->vstart = vtimestamp;
}

syscall:::return
 /self->start/
 {
       this->elapsed = timestamp - self->start;
       this->oncpu = vtimestamp - self->vstart;
```

```
            @Elapsed[probefunc] = sum(this->elapsed);
            @CPU[probefunc] = sum(this->cpu);
            self->start = 0;
            self->vstart = 0;
    }
```

A `self->ok` variable is set beforehand to true if the current process is supposed to be traced. The code is then straightforward: Time stamps are set on the entry to syscalls so that deltas can be calculated on the return.

Examples

Examples include usage and file system archive.

Usage

Command-line options can be listed using `-h`:

```
solaris# procsystime -h
lox# ./procsystime -h
USAGE: procsystime [-aceho] [ -p PID | -n name | command ]
                  -p PID          # examine this PID
                  -n name         # examine this process name
                  -a              # print all details
                  -e              # print elapsed times
                  -c              # print syscall counts
                  -o              # print CPU times
                  -T              # print totals
  eg,
       procsystime -p 1871     # examine PID 1871
       procsystime -n tar      # examine processes called "tar"
       procsystime -aTn bash   # print all details for bash
       procsystime df -h       # run and examine "df -h"
```

File System Archive

The `tar(1)` command was used to archive a file system, with procsystime tracing elapsed times (which is the default) for processes named `tar`:

```
solaris# procsystime -n tar
Tracing... Hit Ctrl-C to end...
^C

Elapsed Times for processes tar,

         SYSCALL          TIME (ns)
           fcntl              58138
         fstat64              96490
          openat             280246
           chdir            1444153
           write            8922505
          open64           15294117
```

continues

```
          openat64           16804949
             close           17855422
        getdents64           46679462
         fstatat64           98011589
              read         1551039139
```

Most of the elapsed time for the `tar(1)` command was in the `read()` syscall, which is expected because `tar(1)` is reading files from disk (which is slow I/O). The total time spent waiting for `read()` syscalls during the procsystime trace was 1.55 seconds.

uoncpu.d

This is a script version of the DTrace one-liner to profile the user stack trace of a given application process name. As one of the most useful one-liners, it may save typing to provide it as a script, where it can also be more easily enhanced.

Script

```
1      #!/usr/sbin/dtrace -s
2
3      profile:::profile-1001
4      /execname == $$1/
5      {
6            @["\n  on-cpu (count @1001hz):", ustack()] = count();
7      }
```

Script uoncpu.d

Example

Here the `uoncpu.d` script is used to frequency count the user stack trace of all currently running Perl programs. Note `perl` is passed as a command-line argument, evaluated in the predicate (line 4):

```
# uoncpu.d perl
dtrace: script 'uoncpu.d' matched 1 probe
^C
[...output truncated...]

  on-cpu (count @1001hz):
            libperl.so.1`Perl_sv_setnv+0xc8
            libperl.so.1`Perl_pp_multiply+0x3fe
            libperl.so.1`Perl_runops_standard+0x3b
            libperl.so.1`S_run_body+0xfa
            libperl.so.1`perl_run+0x1eb
            perl`main+0x8a
            perl`_start+0x7d
            105
```

```
  on-cpu (count @1001hz):
              libperl.so.1`Perl_pp_multiply+0x3f7
              libperl.so.1`Perl_runops_standard+0x3b
              libperl.so.1`S_run_body+0xfa
              libperl.so.1`perl_run+0x1eb
              perl`main+0x8a
              perl`_start+0x7d
              111
```

The hottest stacks identified include the `Perl_pp_multiply()` function, suggesting that Perl is spending most of its time doing multiplications. Further analysis of those functions and using the perl provider, if available (see Chapter 8), could confirm.

uoffcpu.d

As a companion to `uoncpu.d`, the `uoffcpu.d` script measures the time spent off-CPU by user stack trace. This time includes device I/O, lock wait, and dispatcher queue latency.

Script

```
#!/usr/sbin/dtrace -s

sched:::off-cpu
/execname == $$1/
{
      self->start = timestamp;
}

sched:::on-cpu
 /self->start/
 {
        this->delta = (timestamp - self->start) / 1000;
        @["off-cpu (us):", ustack()] = quantize(this->delta);
        self->start = 0;
 }
```

Script uoffcpu.d

Example

Here the `uoffcpu.d` script was used to trace CPU time of bash shell processes:

```
# uoffcpu.d bash
dtrace: script 'uoffcpu.d' matched 6 probes
^C
[...]
```

continues

```
    off-cpu (us):
              libc.so.1`__waitid+0x7
              libc.so.1`waitpid+0x65
              bash`0x8090627
              bash`wait_for+0x1a4
              bash`execute_command_internal+0x6f1
              bash`execute_command+0x5b
              bash`reader_loop+0x1bf
              bash`main+0x7df
              bash`_start+0x7d

           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
         1048576 |                                         0

    off-cpu (us):
              libc.so.1`__read+0x7
              bash`rl_getc+0x47
              bash`rl_read_key+0xeb
              bash`readline_internal_char+0x99
              bash`0x80d945a
              bash`0x80d9481
              bash`readline+0x55
              bash`0x806e11f
              bash`0x806dff4
              bash`0x806ed06
              bash`0x806f9b4
              bash`0x806f3a4
              bash`yyparse+0x4b9
              bash`parse_command+0x80
              bash`read_command+0xd9
              bash`reader_loop+0x147
              bash`main+0x7df
              bash`_start+0x7d

           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@             5
          131072 |@@@@@@@@@@@                               2
          262144 |                                         0
```

While tracing, in another bash shell, the command sleep 1 was typed and executed. The previous output shows the keystroke latency (mostly 65 ms to 131 ms) of the read commands, as well as the time spent waiting for the sleep(1) command to complete (in the 524 to 1048 ms range, which matches expectation: 1000 ms).

Note the user stack frame generated by the ustack() function contains a mix of symbol names and hex values (for example, bash`0x806dff4) in the output. This can happen for one of several reasons whenever ustack() is used. DTrace actually collects and stores the stack frames has hex values. User addresses are resolved to symbol names as a postprocessing step before the output is generated. It is possible DTrace will not be able to resolve a user address to a symbol name if any of the following is true:

- The user process being traced has exited before the processing can be done.

The symbol table has been stripped, either from the user process binary or from the shared object libraries it has linked.

We are executing user code out of data via jump tables.[6]

plockstat

plockstat(1M) is a powerful tool to examine user-level lock events, providing details on contention and hold time. It uses the DTrace plockstat provider, which is available for developing custom user-land lock analysis scripts. The plockstat provider (and the plockstat(1M) tool) is available on Solaris and Mac OS X and is currently being developed for FreeBSD.

Script

plockstat(1M) is a binary executable that dynamically produces a D script that is sent to libdtrace (instead of a static D script sent to libdtrace via dtrace(1M)). If it is of interest, this D script can be examined using the -V option:

```
solaris# plockstat -V -p 12219
plockstat: vvvv D program vvvv
plockstat$target:::rw-block
{
        self->rwblock[arg0] = timestamp;
}
plockstat$target:::mutex-block
{
        self->mtxblock[arg0] = timestamp;
}
plockstat$target:::mutex-spin
{
        self->mtxspin[arg0] = timestamp;
}
plockstat$target:::rw-blocked
/self->rwblock[arg0] && arg1 == 1 && arg2 != 0/
{
        @rw_w_block[arg0, ustack(5)] =
            sum(timestamp - self->rwblock[arg0]);
        @rw_w_block_count[arg0, ustack(5)] = count();
        self->rwblock[arg0] = 0;
        rw_w_block_found = 1;
}
[...output truncated...]
```

Example

Here the plockstat(1M) command traced all lock events (-A for both hold and contention) on the Name Service Cache Daemon (nscd) for 60 seconds:

6. See *www.opensolaris.org/jive/thread.jspa?messageID=436419&#436419*.

```
solaris# plockstat -A -e 60 -p `pgrep nscd`
Mutex hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30  1302583 0x814c08c                    libnsl.so.1`rpc_fd_unlock+0x4d
  326    15687 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x6c
    7   709342 libumem.so.1`umem_cache_lock libumem.so.1`umem_cache_applyall+0x5e
  112    16702 0x80b67b8                    nscd`lookup_int+0x611
    3   570898 0x81a0548                    libscf.so.1`scf_handle_bind+0x231
   60    24592 0x80b20e8                    nscd`_nscd_mutex_unlock+0x8d
   50    24306 0x80b2868                    nscd`_nscd_mutex_unlock+0x8d
   30    19839 libnsl.so.1`_ti_userlock     libnsl.so.1`sig_mutex_unlock+0x1e
    7    83100 libumem.so.1`umem_update_lock libumem.so.1`umem_update_thread+0x129
[...output truncated...]

R/W reader hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30    95341 0x80c0e14                    nscd`_nscd_get+0xb8
  120    15586 nscd`nscd_nsw_state_base_lock nscd`_get_nsw_state_int+0x19c
   60    20256 0x80e0a7c                    nscd`_nscd_get+0xb8
  120     9806 nscd`addrDB_rwlock           nscd`_nscd_is_int_addr+0xd1
   30    39155 0x8145944                    nscd`_nscd_get+0xb8
[...output truncated...]

R/W writer hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30    16293 nscd`addrDB_rwlock           nscd`_nscd_del_int_addr+0xeb
   30    15440 nscd`addrDB_rwlock           nscd`_nscd_add_int_addr+0x9c
    3    14279 nscd`nscd_smf_service_state_lock nscd`query_smf_state+0x17b

Mutex block

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    2   119957 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e

Mutex spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    1    37959 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e

Mutex unsuccessful spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    2    42988 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e
```

While tracing, there were very few contention events and many hold events. Hold events are normal for software execution and are ideally as short as possible, while contention events can cause performance issues as threads are waiting for locks.

The output has caught a spin event for the lock at address `0x8089ab8` (no symbol name) from the code path location `nscd`_nscd_restart_if_cfgfile_changed+0x3e`, which was for 38 us. This means a thread span on-CPU for 38 us before being able to grab the lock. On two other occasions, the thread gave up spinning after an average of 43 us (unsuccessful spin) and was blocked for 120 us (block), both also shown in the output.

kill.d

The `kill.d` script prints details of process signals as they are sent, such as the PID source and destination, signal number, and result. It's named `kill.d` after the `kill()` system call that it traces, which is used by processes to send signals.

Script

This is based on the `kill.d` script from the DTraceToolkit, which uses the syscall provider to trace the `kill()` syscall. The proc provider could also be used via the `signal-*` probes, which will match other signals other than via `kill()` (see `sigdist.d` next).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7          printf("%-6s %12s %6s %-8s %s\n",
8              "FROM", "COMMAND", "SIG", "TO", "RESULT");
9   }
10
11  syscall::kill:entry
12  {
13          self->target = (int)arg0;
14          self->signal = arg1;
15  }
16
17  syscall::kill:return
18  {
19          printf("%-6d %12s %6d %-8d %d\n",
20              pid, execname, self->signal, self->target, (int)arg0);
21          self->target = 0;
22          self->signal = 0;
23  }
```

Script kill.d

Note that the target PID is cast as a signed integer on line 13; this is because the `kill()` syscall can also send signals to process groups by providing the process group ID as a negative number, instead of the PID. By casting it, it will be correctly printed as a signed integer on line 19.

Example

Here the `kill.d` script has traced the `bash` shell sending signal 9 (`SIGKILL`) to PID 12838 and sending signal 2 (`SIGINT`) to itself, which was a Ctrl-C. `kill.d` has also traced `utmpd` sending a 0 signal (the null signal) to various processes: This signal is used to check that PIDs are still valid, without signaling them to do anything (see `kill(2)`).

```
# kill.d
FROM       COMMAND    SIG TO       RESULT
12224         bash      9 12838    0
3728         utmpd      0 4174     0
3728         utmpd      0 3949     0
3728         utmpd      0 10621    0
3728         utmpd      0 12221    0
12224         bash      2 12224    0
```

sigdist.d

The `sigdist.d` script shows which processes are sending which signals to other processes, including the process names. This traces all signals: the `kill()` system call as well as kernel-based signals (for example, alarms).

Script

This script is based on `/usr/demo/dtrace/sig.d` from Oracle Solaris and uses the proc provider `signal-send` probe.

```
#!/usr/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

proc:::signal-send
{
      @Count[execname, stringof(args[1]->pr_fname), args[2]] = count();
}

dtrace:::END
{
      printf("%16s %16s %6s %6s\n", "SENDER", "RECIPIENT", "SIG", "COUNT");
      printa("%16s %16s %6d %6@d\n", @Count);
}
```

Script sigdist.d

Example

The `sigdist.d` script has traced the bash shell sending signal 9 (`SIGKILL`) to a sleep process and also signal 2 (`SIGINT`, Ctrl-C) to itself. It's also picked up `sshd` sending `bash` the `SIGINT`, which happened via a syscall `write()` of the Ctrl-C to the ptm (`STREAMS` pseudo-tty master driver) device for `bash`, not via the `kill()` syscall.

```
# sigdist.d
Tracing... Hit Ctrl-C to end.
^C
          SENDER         RECIPIENT    SIG   COUNT
            bash              bash      2       1
            bash             sleep      9       1
            sshd              bash      2       1
            sshd            dtrace      2       1
           sched              bash     18       2
            bash              bash     20       3
           sched          sendmail     14       3
           sched          sendmail     18       3
           sched           proftpd     14       7
           sched        in.mpathd      14      10
```

threaded.d

The `threaded.d` script provides data for quantifying how well multithreaded applications are performing, in terms of parallel execution across CPUs. If an application has sufficient CPU bound work and is running on a system with multiple CPUs, then ideally the application would have multiple threads running on those CPUs to process the work in parallel.

Script

This is based on the `threaded.d` script from the DTraceToolkit.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

profile:::profile-101
/pid != 0/
{
      @sample[pid, execname] = lquantize(tid, 0, 128, 1);
}

 profile:::tick-1sec
 {
       printf("%Y,\n", walltimestamp);
       printa("\n @101hz   PID: %-8d CMD: %s\n%@d", @sample);
       printf("\n");
       trunc(@sample);
 }
```

Script threaded.d

Example

To demonstrate `threaded.d`, two programs were written (called test0 and test1) that perform work on multiple threads in parallel. One of the programs was coded with a lock "serialization" issue, where only the thread holding the lock can really make forward progress. See whether you can tell which one:

```
# threaded.d
2010 Jul  4 05:17:09,

 @101hz   PID: 12974    CMD: test0

           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@@@@@@@                               28
               3 |@@                                       6
               4 |@@@@@@@@@@@@                             32
               5 |@@@@@                                    14
               6 |@@@@@                                    15
               7 |@@@                                      8
               8 |@@                                       5
               9 |@@@                                      10
              10 |                                         0

 @101hz   PID: 12977    CMD: test1

           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@                                     77
               3 |@@@@@@                                   97
               4 |@@@@                                     77
               5 |@@@@@                                    87
               6 |@@@@                                     76
               7 |@@@@@@                                   101
               8 |@@@@                                     76
               9 |@@@@@@                                   100
              10 |                                         0

[...]
```

`threaded.d` prints output every second, which shows a distribution plot where `value` is the thread ID and `count` is the number of samples during that second. By glancing at the output, both programs had every thread sampled on-CPU during the one second, so the issue may not be clear. The clue is in the counts: `threaded.d` is sampling at 101 Hertz (101 times per second), and the sample counts for test0 only add up to 118 (a little over one second worth of samples on one CPU), whereas test1 adds up to 691. The program with the issue is test0, which is using a fraction of the CPU cycles that test1 is able to consume in the same interval.

This was a simple way to analyze the CPU execution of a multithreaded application. A more sophisticated approach would be to trace kernel scheduling events (the sched provider) as the application threads stepped on- and off-CPU.

Case Studies

In this section, we apply the scripts and methods discussed in this chapter to observe and measure applications with DTrace.

Firefox idle

This case study examines the Mozilla Firefox Web browser version 3, running on Oracle Solaris.

The Problem

Firefox is 8.9 percent on-CPU yet has not been used for hours. What is costing 8.9 percent CPU?

```
# prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 27060 brendan   856M  668M sleep   59    0   7:30:44 8.9% firefox-bin/17
 27035 brendan   150M  136M sleep   59    0   0:20:51 0.4% opera/3
 18722 brendan   164M   38M sleep   59    0   0:57:53 0.1% java/18
  1748 brendan  6396K 4936K sleep   59    0   0:03:13 0.1% screen-4.0.2/1
 17303 brendan   305M  247M sleep   59    0  34:16:57 0.1% Xorg/1
 27754 brendan  9564K 3772K sleep   59    0   0:00:00 0.0% sshd/1
 19998 brendan    68M 7008K sleep   59    0   2:41:34 0.0% gnome-netstatus/1
 27871 root     3360K 2792K cpu0    49    0   0:00:00 0.0% prstat/1
 29805 brendan    54M   46M sleep   59    0   1:53:23 0.0% elinks/1
[...]
```

Profiling User Stacks

The `uoncpu.d` script (from the "Scripts" section) was run for ten seconds:

```
# uoncpu.d firefox-bin
dtrace: script 'uoncpu.d' matched 1 probe
^C
[...output truncated...]

  on-cpu (count @1001hz):
              libmozjs.so`js_FlushPropertyCacheForScript+0xe6
              libmozjs.so`js_DestroyScript+0xc1
              libmozjs.so`JS_EvaluateUCScriptForPrincipals+0x87
              libxul.so`__1cLnsJSContextOEvaluateString6MrknSnsAString_internal_pvpnMn
sIPrincip8
              libxul.so`__1cOnsGlobalWindowKRunTimeout6MpnJnsTimeout__v_+0x59c
              libxul.so`__1cOnsGlobalWindowNTimerCallback6FpnInsITimer_pv_v_+0x2e
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x144
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
```
continues

```
              libxul.so`__1cMnsAppStartupDRun6M_I_+0x34
              libxul.so`XRE_main+0x35e3
              firefox-bin`main+0x223
              firefox-bin`_start+0x7d
               42
```

The output was many pages long and includes C++ signatures as function names (they can be passed through `c++filt` to improve readability). The hottest stack is in libmozjs, which is SpiderMonkey—the Firefox JavaScript engine. However, the count for this hot stack is only 42, which, when the other counts from the numerous truncated pages are tallied, is likely to represent only a fraction of the CPU cycles. (`uoncpu.d` can be enhanced to print a total sample count and the end to make this ratio calculation easy to do.)

Profiling User Modules

Perhaps an easier way to find the origin of the CPU usage is to not aggregate on the entire user stack track but just the top-level user module. This won't be as accurate—a user module may be consuming CPU by calling functions from a generic library such as libc—but it is worth a try:

```
# dtrace -n 'profile-1001 /execname == "firefox-bin"/ { @[umod(arg1)] = count(); }
tick-60sec { exit(0); }'
dtrace: description 'profile-1001 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  63284                      :tick-60sec

  libsqlite3.so                                                     1
  0xf0800000                                                        2
  libplds4.so                                                       2
  libORBit-2.so.0.0.0                                               5
  0xf1600000                                                        8
  libgthread-2.0.so.0.1400.4                                       10
  libgdk-x11-2.0.so.0.1200.3                                       14
  libplc4.so                                                       16
  libm.so.2                                                        19
  libX11.so.4                                                      50
  libnspr4.so                                                     314
  libglib-2.0.so.0.1400.4                                         527
  0x0                                                             533
  libflashplayer.so                                              1143
  libc.so.1                                                      1444
  libmozjs.so                                                    2671
  libxul.so                                                      4143
```

The hottest module was libxul, which is the core Firefox library. The next was libmozjs (JavaScript) and then libc (generic system library). It is possible that libmozjs is responsible for the CPU time in both libc and libxul, by calling functions from them. We'll investigate libmozjs (JavaScript) first; if this turns out to be a dead end, we'll return to libxul.

Function Counts and Stacks

To investigate JavaScript, the DTrace JavaScript provider can be used (see Chapter 8). For the purposes of this case study, let's assume that such a convenient provider is not available. To understand what the libmosjs library is doing, we'll first frequency count function calls:

```
# dtrace -n 'pid$target:libmozjs::entry { @[probefunc] = count(); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target:libmozjs::entry ' matched 1617 probes
^C

  CloseNativeIterators                                                1
  DestroyGCArenas                                                     1
  JS_CompareValues                                                    1
  JS_DefineElement                                                    1
  JS_FloorLog2                                                        1
  JS_GC                                                               1
[...]
  JS_free                                                         90312
  js_IsAboutToBeFinalized                                         92414
  js_GetToken                                                     99666
  JS_DHashTableOperate                                           102908
  GetChar                                                        109323
  fun_trace                                                      132924
  JS_GetPrivate                                                  197322
  js_TraceObject                                                 213983
  JS_TraceChildren                                               228323
  js_SearchScope                                                 267826
  js_TraceScopeProperty                                          505450
  JS_CallTracer                                                 1923784
```

The most frequent function called was `JS_CallTracer()`, which was called almost two million times during the ten seconds that this one-liner was tracing. To see what it does, the source code could be examined; but before we do that, we can get more information from DTrace including frequency counting the user stack trace to see who is calling this function:

```
# dtrace -n 'pid$target:libmozjs:JS_CallTracer:entry { @[ustack()] =
count(); }' -p `pgrep firefox-bin`
[...]

              libmozjs.so`JS_CallTracer
              libmozjs.so`js_TraceScopeProperty+0x54
              libmozjs.so`js_TraceObject+0xd5
              libmozjs.so`JS_TraceChildren+0x351
              libxul.so`__1cLnsXPConnectITraverse6MpvrnbInsCycleCollectionTraversalCal
lback__I_+0xc7
              libxul.so`__1cQnsCycleCollectorJMarkRoots6MrnOGCGraphBuilder__v_+0x96
              libxul.so`__1cQnsCycleCollectorPBeginCollection6M_i_+0xf1
              libxul.so`__1cbGnsCycleCollector_beginCollection6F_i_+0x26
              libxul.so`__1cZXPCCycleCollectGCCallback6FpnJJSContext_nKJSGCStatus__i_+0xd8
              libmozjs.so`js_GC+0x5ef
```

continues

```
              libmozjs.so`JS_GC+0x4e
              libxul.so`__1cLnsXPConnectHCollect6M_i_+0xaf
              libxul.so`__1cQnsCycleCollectorHCollect6MI_I_+0xee
              libxul.so`__1cYnsCycleCollector_collect6F_I_+0x28
              libxul.so`__1cLnsJSContextGNotify6MpnInsITimer__I_+0x375
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x12d
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
            40190
```

The stack trace here has been truncated (increase the `ustackframes` tunable to see all); however, enough has been seen for this and the truncated stack traces to see that they originate from `JS_GC()`—a quick look at the code confirms that this is JavaScript Garbage Collect.

Function CPU Time

Given the name of the garbage collect function, a script can be quickly written to check the CPU time spent in it (named `jsgc.d`):

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

pid$target::JS_GC:entry
{
        self->vstart = vtimestamp;
}

pid$target::JS_GC:return
/self->vstart/
{
        this->oncpu = (vtimestamp - self->vstart) / 1000000;
        printf("%Y GC: %d CPU ms\n", walltimestamp, this->oncpu);
        self->vstart = 0;
}
```

Script jsgc.d

This specifically measures the elapsed CPU time (vtimestamp) for `JS_GC()`. (Another approach would be to use the profile provider and count stack traces that included `JS_GC()`.)

Here we execute `jsgc.d`:

```
# jsgc.d -p `pgrep firefox-bin`
2010 Jul  4 01:06:57 GC: 331 CPU ms
2010 Jul  4 01:07:38 GC: 316 CPU ms
2010 Jul  4 01:08:18 GC: 315 CPU ms
^C
```

So, although GC is on-CPU for a significant time, more than 300 ms per call, it's not happening frequently enough to explain the 9 percent CPU average of Firefox. This may be a problem, but it's not the problem. (This is included here for completeness; this is the exact approach used to study this issue.)

Another frequently called function was `js_SearchScope()`. Checking its stack trace is also worth a look:

```
# dtrace -n 'pid$target:libmozjs:js_SearchScope:entry { @[ustack()] =
count(); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target:libmozjs:js_SearchScope:entry ' matched 1 probe
^C
[...output truncated...]

              libmozjs.so`js_SearchScope
              libmozjs.so`js_DefineNativeProperty+0x2f1
              libmozjs.so`call_resolve+0x1e7
              libmozjs.so`js_LookupProperty+0x3d3
              libmozjs.so`js_PutCallObject+0x164
              libmozjs.so`js_Interpret+0x9cd4
              libmozjs.so`js_Execute+0x3b4
              libmozjs.so`JS_EvaluateUCScriptForPrincipals+0x58
              libxul.so`__1cLnsJSContextOEvaluateString6MrknSnsAString_internal_pvpnMn
sIPrincipal_pkcIIp1pi_I_+0x2e8
              libxul.so`__1cOnsGlobalWindowKRunTimeout6MpnJnsTimeout__v_+0x59c
              libxul.so`__1cOnsGlobalWindowNTimerCallback6FpnInsITimer_pv_v_+0x2e
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x144
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
              libxul.so`__1cMnsAppStartupDRun6M_I_+0x34
              libxul.so`XRE_main+0x35e3
              firefox-bin`main+0x223
              firefox-bin`_start+0x7d
             9287
```

This time, the function is being called by `js_Execute()`, the entry point for JavaScript code execution (and itself was called by `JS_EvaluateUCScriptForPrincipals()`). Here we are modifying the earlier script to examine on-CPU time (now `jsexecute.d`):

```
#!/usr/sbin/dtrace -s

pid$target::js_Execute:entry
{
        self->vstart = vtimestamp;
}

pid$target::js_Execute:return
/self->vstart/
{
        this->oncpu = vtimestamp - self->vstart;
        @["js_Execute Total(ns):"] = sum(this->oncpu);
        self->vstart = 0;
}
```

Script jsexecute.d

Here we run it for ten seconds:

```
# jsexecute.d -p `pgrep firefox-bin` -n 'tick-10sec { exit(0); }'
dtrace: script 'jsexecute.d' matched 2 probes
dtrace: description 'tick-10sec ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  64907                       :tick-10sec

  js_Execute Total(ns):                                        427936779
```

This shows 428 ms of time in `js_Execute()` during those ten seconds, and so this CPU cost can explain about half of the Firefox CPU time (this is a single-CPU system; therefore, there is 10,000 ms of available CPU time every 10 seconds, so this is about 4.3 percent of CPU).

The JavaScript functions could be further examined with DTrace to find out why this JavaScript program is hot on-CPU, in other words, what exactly it is doing (the DTrace JavaScript provider would help here, or a Firefox add-on could be tried).

Fetching Context

Here we will find what is being executed: preferably the URL. Examining the earlier stack trace along with the Firefox source (which is publically available) showed the JavaScript filename is the sixth argument to the `JS_EvaluateUCScriptForPrincipals()` function. Here we are pulling this in and frequency counting:

```
# dtrace -n 'pid$target::*EvaluateUCScriptForPrincipals*:entry { @[copyinstr(arg5)] =
 count(); } tick-10sec { exit(0); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target::*EvaluateUCScriptForPrincipals*:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  64907                       :tick-10sec

  http://www.example.com/js/st188.js                                         7056
```

The name of the URL has been modified in this output (to avoid embarrassing anyone); it pointed to a site that I didn't think I was using, yet their script was getting executed more than 700 times per second anyway, which is consuming (wasting!) at least 4 percent of the CPU on this system.

The Fix

An add-on was already available that could help at this point: SaveMemory, which allows browser tabs to be paused. The DTrace one-liner was modified to print continual one-second summaries, while all tabs were paused as an experiment:

```
# dtrace -n 'pid$target::*EvaluateUCScriptForPrincipals*:entry { @[copyinstr(arg5)] =
 count(); } tick-1sec { printa(@); trunc(@); }' -p `pgrep firefox-bin`
[...]
  1  63140                       :tick-1sec
  http://www.example.com/js/st188.js                                        697

  1  63140                       :tick-1sec
  http://www.example.com/js/st188.js                                        703

  1  63140                       :tick-1sec
file:///export/home/brendan/.mozilla/firefox/3c8k4kh0.default/extensions/%7Be4a8a97b-f
2ed-450b-b12d-ee082ba24781%7D/components/greasemonkey.js                  1
  http://www.example.com/js/st188.js                                        126


  1  63140                       :tick-1sec

  1  63140                       :tick-1sec
```

The execution count for the JavaScript program begins at around 700 executions per second and then vanishes when pausing all tabs. (The output has also caught the execution of `greasemonkey.js`, executed as the add-on was used.)

`prstat(1M)` shows the CPU problem is no longer there (shown after waiting a few minutes for the %CPU decayed average to settle):

```
# prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 27035 brendan   150M  136M sleep   49    0   0:27:15 0.2% opera/4
 27060 brendan   407M  304M sleep   59    0   7:35:12 0.1% firefox-bin/17
 28424 root     3392K 2824K cpu1    49    0   0:00:00 0.0% prstat/1
[...]
```

Next, the browser tabs were unpaused one by one to identify the culprit, while still running the DTrace one-liner to track JavaScript execution by file. This showed that there were seven tabs open on the same Web site that was running the JavaScript program—each of them executing it about 100 times per second. The Web site is a popular blogging platform, and the JavaScript was being executed by what appears to be an inert icon that links to a different Web site (but as we found out—it is not inert).[7] The exact operation of that JavaScript program can now be investigated using the DTrace JavaScript provider or a Firefox add-on debugger.

Conclusion

A large component of this issue turned out to be a rogue JavaScript program, an issue that could also have been identified with Firefox add-ons. The advantage of

7. An e-mail was sent to the administrators of the blogging platform to let them know.

using DTrace is that if there is an issue, the root cause can be identified—no matter where it lives in the software stack. As an example of this,[8] about a year ago a performance issue was identified in Firefox and investigated in the same way—and found to be a bug in a kernel frame buffer driver (video driver); this would be extremely difficult to have identified from the application layer alone.

Xvnc

Xvnc is a Virtual Network Computing (VNC) server that allows remote access to X server–based desktops. This case study represents examining an Xvnc process that is CPU-bound and demonstrates using the syscall and profile providers.

When performing a routine check of running processes on a Solaris system by using `prstat(1)`, it was discovered that an Xvnc process was the top CPU consumer. Looking just at that process yields the following:

```
solaris# prstat -c -Lmp 5459
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  5459 nobody    86  14 0.0 0.0 0.0 0.0 0.0 0.0   0  36 .2M 166 Xvnc/1
```

We can see the Xvnc process is spending most of its time executing in user mode (USR, 86 percent) and some of its time in the kernel (SYS, 14 percent). Also worth noting is it is executing about 200,000 system calls per second (SCL value of `.2M`).

syscall Provider

Let's start by checking what those system calls are. This one-liner uses the syscall provider to frequency count system calls for this process and prints a summary every second:

```
solaris# dtrace -qn 'syscall:::entry /pid == 5459/ { @[probefunc] =
count(); } tick-1sec { printa(@); trunc(@); }'

  read                                                              4
  lwp_sigmask                                                      34
  setcontext                                                       34
  setitimer                                                        68
  accept                                                        48439
  gtime                                                         48439
  pollsys                                                       48440
  write                                                         97382
```

continues

8. I'd include this as a case study here, if I had thought to save the DTrace output at the time.

```
  read                                                               4
  lwp_sigmask                                                       33
  setcontext                                                        33
  setitimer                                                         66
  gtime                                                          48307
  pollsys                                                        48307
  accept                                                         48308
  write                                                          97117
```

Because the rate of system calls was relatively high, as reported by `prstat(1M)`, we opted to display per-second rates with DTrace. The output shows more than 97,000 `write()` system calls per second and just more than 48,000 `accept()`, `poll()`, and `gtime()` calls.

Let's take a look at the target of all the writes and the requested number of bytes to write:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459/ { @[fds[arg0].fi_pathname,
arg2] = count(); }'
^C

  /var/adm/X2msgs                                                26              8
  /devices/pseudo/mm@0:null                                    8192           3752
  /var/adm/X2msgs                                                82         361594
  /var/adm/X2msgs                                                35         361595
```

The vast majority of the writes are to a file, `/var/adm/X2msgs`. The number of bytes to write was 82 bytes and 35 bytes for the most part (more than 361,000 times each). Checking that file yields the following:

```
solaris# ls -l /var/adm/X2msgs
-rw-r--r--   1 root     nobody   2147483647 Aug 13 15:05 /var/adm/X2msgs
solaris# tail /var/adm/X2msgs
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
```

Looking at the file Xvnc is writing to, we can see it is getting very large (more than 2GB), and the messages themselves appear to be error messages. We will explore that more closely in just a minute.

Given the rate of 97,000 writes per second, we can already extrapolate that each write is taking much less than 1 ms (1/97000 = 0.000010), so we know the data is probably being written to main memory (since the file resides on a file system and the writes are not synchronous, they are being satisfied by the in-memory file system cache). We can of course time these writes with DTrace:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459/
{ @[fds[arg0].fi_fs] = count(); }'
^C
  specfs                                                              2766
  zfs                                                               533090

solaris# cat -n w.d
1   #!/usr/sbin/dtrace -qs
2
3   syscall::write:entry
4   /pid == 5459 && fds[arg0].fi_fs == "zfs"/
5   {
6           self->st = timestamp;
7   }
8   syscall::write:return
9   /self->st/
10  {
11          @ = quantize(timestamp - self->st);
12          self->st = 0;
13  }

solaris# ./w.d
^C

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1477349
            1024 |                                         2312
            2048 |                                         3100
            4096 |                                         250
            8192 |                                         233
           16384 |                                         145
           32768 |                                         90
           65536 |                                         0
```

Before measuring the write time, we wanted to be sure we knew the target file system type of the file being written, which was ZFS. We used that in the predicate in the `w.d` script to measure write system calls for this process (along with the process PID test). The output of `w.d` is a quantize aggregation that displays wall clock time for all the write calls executed to a ZFS file system from that process during the sampling period. We see that most of the writes fall in the 512-nanosecond to 1024-nanosecond range, so these are most certainly writes to memory.

We can determine the user code path leading up to the writes by aggregating on the user stack when the write system call is called:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459 && fds[arg0].fi_fs ==
"zfs"/ { @[ustack()] = count(); }'
^C
[...]
```

```
              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1a5
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           430879

              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1eb
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           430879
```

We see two very similar stack frames, indicating a log event is causing the Xvnc process to write to its log file.

We can even use DTrace to observe what is being written to the file, by examining the contents of the buffer pointer from the `write(2)` system call. It is passed to the `copyinstr()` function, both to copy the data from user-land into the kernel address space and to treat it as a string:

```
solaris# dtrace -n 'syscall::write:entry /pid == 5459/ { @[copyinstr(arg1)] =
count(); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

Sun Aug 22 00:09:05 2010
ent (22)
keupHandler: unable to accept new
             st!
Ltd.
See http://www.realvnc.com for information on VNC.
                1

Sun Aug 22 00:09:06 2010
ent (22)
keupHandler: unable to accept new
             st!
                2
[...]
upHandler: unable to accept new connection: Invalid argument (22)XserverDesktop::wakeu
pHandler: unable to accept new connection: Invalid argument (22)XserverDesktop::wakeup
Handler: unable to accept new connection: Invalid argument (22)XserverDesktop::wake
```

```
                59
valid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: I
nvalid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: In
valid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: In
                59
```

This shows the text being written to the log file, which largely contains errors describing invalid arguments used for new connections. Remember that our initial one-liner discovered more than 48,000 `accept()` system calls per-second—it would appear that these are failing because of invalid arguments, which is being written as an error message to the `/var/adm/X2msgs` log.

DTrace can confirm that the `accept()` system calls are failing in this way, by examining the error number (errno) on syscall return:

```
solaris# dtrace -n 'syscall::accept:return /pid == 5459/ { @[errno] = count(); }'
dtrace: description 'syscall::accept:return ' matched 1 probe
^C

       22           566135

solaris# grep 22 /usr/include/sys/errno.h
#define     EINVAL      22    /* Invalid argument                 */
```

All the `accept()` system calls are returning with errno 22, `EINVAL` (Invalid argument). The reason for this can be investigated by examining the arguments to the `accept()` system call.

```
solaris# dtrace -n 'syscall::accept:entry /execname == "Xvnc"/ { @[arg0, arg1,
arg2] = count(); }'
dtrace: description 'syscall::accept:entry ' matched 1 probe
^C

                3                0                0           150059
```

We see the first argument to accept is 3, which is the file descriptor for the socket. The second two arguments are both `NULL`, which *may* be the cause of the `EINVAL` error return from accept. It is possible it is valid to call `accept` with the second and third arguments as `NULL` values,[9] in which case the Xvnc code is not handling the error return properly. In either case, the next step would be to look at the Xvnc source code and find the problem. The code is burning a lot of CPU with calls to `accept(2)` that are returning an error and each time generating a log file write.

9. Stevens (1998) indicates that it is.

While still using the syscall provider, the user code path for another of the other hot system calls can be examined:

```
solaris# dtrace -n 'syscall::gtime:entry /pid == 5459/ { @[ustack()] = count(); }'
dtrace: description 'syscall::gtime:entry ' matched 1 probe
^C

              libc.so.1`__time+0x7
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0xce
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           370156
```

This shows that calls to `gtime(2)` are part of the log file writes in the application, based on the user function names we see in the stack frames.

profile Provider

To further understand the performance of this process, we will sample the on-CPU code at a certain frequency, using the profile provider.

```
solaris# dtrace -n 'profile-997hz /arg1 && pid == 5459/ { @[ufunc(arg1)] = count(); }'
dtrace: description 'profile-997hz ' matched 1 probe
^C
[...]
  libc.so.1`memcpy                                                  905
  Xvnc`_ZN14XserverDesktop12blockHandlerEP6fd_set                   957
  libgcc_s.so.1`uw_update_context_1                                1155
  Xvnc`_ZN3rdr15SystemExceptionC2EPKci                             1205
  libgcc_s.so.1`execute_cfa_program                                1278
  libc.so.1`strncat                                                1418
  libc.so.1`pselect                                                1686
  libstdc++.so.6.0.3`_Z12read_uleb128PKhPj                         1700
  libstdc++.so.6.0.3`_Z28read_encoded_value_with_basehjPKhPj       2198
  libstdc++.so.6.0.3`__gxx_personality_v0                          2445
  libc.so.1`_ndoprnt                                               3918
```

This one-liner shows which user functions were on-CPU most frequently. It tests for user mode (`arg1`) and the process of interest and uses the `ufunc()` function to convert the user-mode on-CPU program counter (`arg1`) into the user function name. The most frequent is a libc function, `_ndoprnt()`, followed by several functions from the standard C++ library.

For a detailed look of the user-land code path that is responsible for consuming CPU cycles, aggregate on the user stack:

```
solaris# dtrace -n 'profile-997hz /arg1 && pid == 5459/ { @[ustack()] =
count(); } tick-10sec { trunc(@, 20); exit(0); }'
^c
[...]
              libstdc++.so.6.0.3`__gxx_personality_v0+0x29f
              libgcc_s.so.1`_Unwind_RaiseException+0x88
              libstdc++.so.6.0.3`__cxa_throw+0x64
              Xvnc`_ZN7network11TcpListener6acceptEv+0xb3
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              125

              libc.so.1`memset+0x10c
              libgcc_s.so.1`_Unwind_RaiseException+0xb7
              libstdc++.so.6.0.3`__cxa_throw+0x64
              Xvnc`_ZN7network11TcpListener6acceptEv+0xb3
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              213
```

Note that only the two most frequent stack frames are shown here. We see the event loop in the Xvnc code and visually decoding the mangled function names; we can see a function with `network TCPListener accept` in the function name. This makes sense for an application like Xvnc, which would be listening on a network socket for incoming requests and data. And we know that there's an issue with the issued `accept(2)` calls inducing a lot of looping around with the error returns.

We can also take a look at the kernel component of the CPU cycles consumed by this process, again using the profile provider and aggregating on kernel stacks:

```
solaris# dtrace -n 'profile-997hz /pid == 5459 && arg0/ { @[stack()] = count(); }'
^c
[...]
              unix`mutex_enter+0x10
              genunix`pcache_poll+0x1a5
              genunix`poll_common+0x27f
              genunix`pollsys+0xbe
              unix`sys_syscall32+0x101
               31

              unix`tsc_read+0x3
              genunix`gethrtime+0xa
              unix`pc_gethrestime+0x31
              genunix`gethrestime+0xa
              unix`gethrestime_sec+0x11
              genunix`gtime+0x9
```

```
              unix`sys_syscall32+0x101
               41

              unix`tsc_read+0x3
              genunix`gethrtime_unscaled+0xa
              genunix`syscall_mstate+0x4f
              unix`sys_syscall32+0x11d
              111

              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0
              229
```

The kernel stack is consistent with previously observed data. We see system call processing (remember, this process is doing 200,000 system calls per second), we see the gtime system call stack in the kernel, as well as the poll system call kernel stack. We could measure this to get more detail, but the process profile was only 14 percent kernel time, and given the rate and type of system calls being executed by this process, there is minimal additional value in terms of understanding the CPU consumption by this process in measuring kernel functions.

For a more connected view, we can trace code flow from user mode through the kernel by aggregating on both stacks:

```
solaris# dtrace -n 'profile-997hz /pid == 5459/ { @[stack(), ustack()] =
count(); } tick-10sec { trunc(@, 2); exit(0); }'
dtrace: description 'profile-997hz ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1 122538                      :tick-10sec


              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0

              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1eb
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              211

              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0
```

continues

```
                libc.so.1`_so_accept+0x7
                Xvnc`_ZN7network11TcpListener6acceptEv+0x18
                Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
                Xvnc`vncWakeupHandler+0x3d
                Xvnc`WakeupHandler+0x36
                Xvnc`WaitForSomething+0x28d
                Xvnc`Dispatch+0x76
                Xvnc`main+0x3e5
                Xvnc`_start+0x80
                493
```

Here we see the event loop calling into the `accept(3S)` interface in libc and entering the system call entry point in the kernel. The second set of stack frames shows the log write path. One of the stacks has also caught `_ndoprnt`, which we know from earlier to be the hottest on-CPU function, calling `write()` as part of Xvnc logging.

Conclusions

The initial analysis with standard operating system tools showed that the single-threaded Xvnc process was CPU bound, spending most of its CPU cycles in user-mode and performing more than 200,000 system calls per second. DTrace was used to discover that the application was continually encountering new connection failures because of invalid arguments (`accept(2)`) and was writing this message to a log file, thousands of times per second.

Summary

With DTrace, applications can be studied like never before: following the flow of code from the application source, through libraries, through system calls, and through the kernel. This chapter completed the topics for application analysis; see other chapters in this book for related topics, including the analysis of programming languages, disk, file system, and network I/O.

10

Databases

DTrace is a powerful tool for analyzing databases, allowing database operation to be examined in detail. High-level events such as user connections and transactions can be traced as they are processed step-by-step inside the database engine and as the database interacts with the operating system. You can get answers to questions such as the following.

- Which queries are causing random disk I/O?
- What is causing high latency for certain transactions?
- How well do the caches perform for certain queries?

Databases already provide a collection of fixed statistics, often extensive and covering details such as transaction times, byte counts, and cache hit rates. Although these are useful, they are limited to what the database makes available and are also limited to one layer of the software stack: the database itself. However, when you need to look outside this layer, for example, at disk events using tools from the operating system, database context is lost. Because DTrace can monitor both types of events in the same tool, it can be used to associate database context to system events. This is why DTrace can be used to determine which queries are causing random disk I/O, by tracing both database and disk I/O events in the same script.

The aim of this chapter is to introduce what you can do with DTrace, to suggest a strategy for analysis, to list DTrace providers, and to provide example DTrace one-liners and scripts for retrieving database context such as query strings. These examples can be combined with those from other chapters to see operating system events such as disk and network I/O in the context of queries and other database events.

Capabilities

The following summarizes DTrace's capabilities on the **database server**.

DTrace provides custom, high-level observability. Information such as query counts and user connections can be traced, and you can decide how to present the results. Although similar information is usually available via other database tools, DTrace provides data-processing features such as frequency counts, distribution plots, and predicates, which can improve how this data is presented and understood.

DTrace can measure behavior across the entire software stack. This can reveal issues with the disk devices, kernel drivers, user-level locks, and any other system component, which might be missed when performing analysis from the database only. Systemic bottlenecks can be identified and eliminated.

DTrace can monitor system events in database context. System events such as disk and network I/O can be analyzed in terms of database users and queries.

DTrace can provide a deep view of internal database operations, going far beyond shipped metrics and standard analyzers.

Apart from the database server, DTrace can also analyze the database clients and their application software, if DTrace is available on those systems.

The following summarizes the capabilities on the **database client**.

DTrace can trace connections to the database server. The database request latency, plus the operation of the database interface library and the client network stack, can be measured from the client.

DTrace can trace the client application. Database transactions can be traced right back to application context. This could lead to changes in the application configuration or code to optimize how database requests are made.

To visualize key components involved, Figure 10-1 shows a database client on a different system from the server.

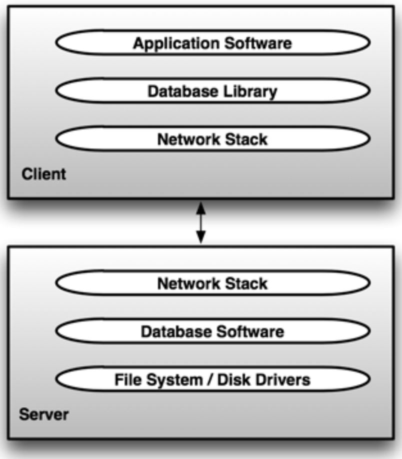


Figure 10-1 Client-server components

Although database administrators can become skilled at identifying issues within the database itself, this is only one component of a larger system. Since DTrace can examine this entire system, the problem is no longer getting to the information but knowing where to start.

Strategy

We suggest the following approach to getting started with database analysis:

1. Use the shipped **database statistics** and analyzers already available. Most databases can provide exhaustive statistics, if enabled, and also analyzers to observe operation. Understanding your available database statistics is a good introduction to which type of metrics may be important and should lead to ideas on where DTrace can *extend* observability (rather than reinvent observability).
2. Search for a matching DTrace **database provider**. This will be a high-level provider that will allow you to write concise scripts to observe fundamental database behavior, for example, the `mysqld` provider.
3. DTrace system behavior from **other providers**. DTrace is about observing the entire system, not just the database. Some issues may be easier to see, and much easier to prove, if observed from other layers of the software stack.

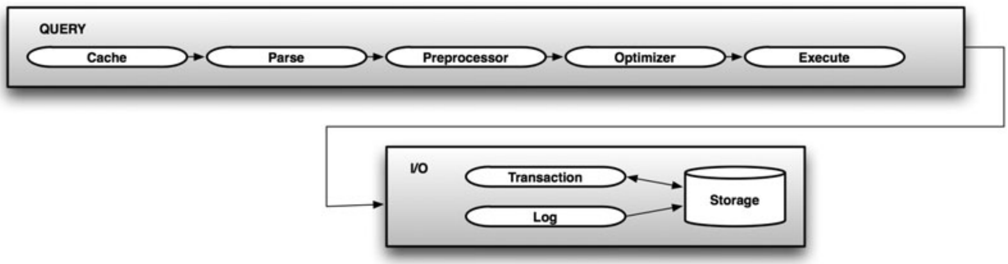


Figure 10-2 Generic database query processing

For example, try observing disk events from the `io` provider (which resembles physical calls to disks), rather than from database I/O.

4. Consider using the `pid` **provider** to examine raw database operation. This can be difficult, because you are observing the raw, unvarnished code of the database in flight. A specific database provider is preferable, but one may not yet have been written for your database. Also consider whether your database vendor allows you to examine the operation of its code (conditions of use).

In general, start tracing at a high level, such as of what work is being performed, and then drill down into the specifics of *how* it is being performed. You may find the biggest wins in eliminating unnecessary work, rather than tuning existing work. Try to picture the key components of the database in your environment and the database engine.

For example, consider Figure 10-2, which shows a generic database query processing engine.

Consider how many of these components you can currently observe. Writing DTrace scripts to trace transactions at each of these components is a good starting point. Scripts can then be enhanced to measure exactly how transactions are being processed.

Providers

Various providers are available for database analysis, with more being added over time. Tables 10-1 and 10-2 list what is currently available for both the database server and client.

The pid provider is considered an “unstable” interface, because it instruments a specific software version. The pid provider–based one-liners and scripts in this chapter may not execute without modifications to match the software version you are running. See Chapter 9, Applications, for more discussion about the pid provider.

Table 10-1 Database Server Providers

Provider	Description
mysql	MySQL database provider. High-level probes for connections, query events, row operations, network I/O, and more.
postgresql	PostgreSQL database provider. High-level probes for transactions and lock events.
syscall	Trace interface between database server and operating system.
io	Trace server disk events: size, latency, throughput.
fbt	Trace kernel events including networking.
pid	Trace server internals: all software function calls.

Table 10-2 Database Client Providers

Provider	Description
syscall	Traces interface between database client and operating system
fbt	Traces kernel events including networking
pid	Traces database client library calls and application software calls

MySQL

A DTrace provider called *mysql* has been developed and added to the MySQL[1] server source. An early version of this provider was in MySQL 6.0.8 and provided a limited set of probes when compiled with the `--enable-dtrace` option. An extended version of the provider was developed and made available in MySQL 6.0.8 and 5.1.30, which is demonstrated here. The probes for the mysql provider are fully documented in the MySQL Reference Manual.[2] Note the probes listed in Table 10-3 are from the MySQL 5.7.1 Reference Manual.

Without the mysql provider, similar functionality is possible by tracing using the pid provider, although such scripts will be unstable and require updates to continue working on different versions of MySQL. They will also require an understanding of the MySQL server source code to develop.

One-liners and scripts are demonstrated here as an introduction to DTrace and MySQL.

1. *www.mysql.com*
2. *http://dev.mysql.com/doc/refman/5.6/en/dba-dtrace-mysqld-ref.html*

Table 10-3 MySQL DTrace Probes

Probe Group	Probe Name
Connection	`connection-start`, `connection-done`
Command	`command-start`, `command-done`
Query	`query-start`, `query-done`
Query parsing	`query-parse-start`, `query-parse-done`
Query cache	`query-cache-hit`, `query-cache-miss`
Query execution	`query-exec-start`, `query-exec-done`
Row level	`insert-row-start`, `insert-row-done`, `update-row-start`, `update-row-done`, `delete-row-start`, `delete-row-done`
Row reads	`read-row-start`, `read-row-done`
Index reads	`index-read-row-start`, `index-read-row-done`
Lock	`handler-rdlock-start`, `handler-rdlock-done`, `handler-wrlock-start`, `handler-wrlock-done`, `handler-unlock-start`, `handler-unlock-done`
Filesort	`filesort-start`, `filesort-done`
Statement	`select-start`, `select-done`, `insert-start`, `insert-done`, `insert-select-start`, `insert-select-done`, `update-start`, `update-done`, `multi-update-start`, `multi-update-done`, `delete-start`, `delete-done`, `multi-delete-start`, `multi-delete-done`
Network	`net-read-start`, `net-read-done`, `net-write-start`, `net-write-done`
Keycache	`keycache-read-start`, `keycache-read-block`, `keycache-read-done`, `keycache-read-hit`, `keycache-read-miss`, `keycache-write-start`, `keycache-write-block`, `keycache-write-done`

One-Liners

The following one-liners provide an excellent starting point for observing and understanding database activity.

mysql Provider

MySQL: query trace by query string:

```
dtrace -n 'mysql*:::query-start { trace(copyinstr(arg0)) }'
```

MySQL: query count summary by query string:

```
dtrace -n 'mysql*:::query-start { @[copyinstr(arg0)] = count(); }'
```

MySQL: query count summary by user:

```
dtrace -n 'mysql*:::query-start { @[copyinstr(arg3)] = count(); }'
```

MySQL: query count summary by host:

```
dtrace -n 'mysql*:::query-start { @[copyinstr(arg4)] = count(); }'
```

MySQL: query event count:

```
dtrace -n 'mysql*:::query-* { @[probename] = count(); }'
```

MySQL: row event count:

```
dtrace -n 'mysql*:::*-row-* { @[probename] = count(); }'
```

MySQL: lock event count:

```
dtrace -n 'mysql*:::*lock-* { @[probename] = count(); }'
```

pid Provider

The following are examples of tracing MySQL 5.1 internal functions using the pid provider; these are likely to need the function name and arguments adjusted for other versions.

MySQL server: trace queries:

```
dtrace -qn 'pid$target::*mysql_parse*:entry { printf("%Y   %s\n", walltimestamp,
copyinstr(arg1)); }' -p PID
```

MySQL server: count queries:

```
dtrace -n 'pid$target::*mysql_parse*:entry { @[copyinstr(arg1)] = count(); }' -p PID
```

MySQL client: who's doing what (stack trace by query):

```
dtrace -Zn 'pid$target:libmysql*:mysql_*query:entry { trace(copyinstr(arg1));
ustack(); }' -p PID
```

io Provider

MySQL: disk I/O size distribution

```
dtrace -n 'io:::start /execname == "mysqld"/ { @ = quantize(args[0]->b_bcount); }'
```

One-Liner Selected Examples

This section includes several of the one-liners from the previous section in action.

MySQL: Query Count Summary by Query String

This one-liner counts queries while tracing on the MySQL server:

```
server# dtrace -n 'mysql*:::query-start { @[copyinstr(arg0)] = count(); }'
dtrace: description 'mysql*:::query-start ' matched 1 probe
^C

  select * from imagelinks                                          1
  select * from imagelinks where il_from > 118                      1
  select * from user                                                1
  select @@version_comment limit 1                                  1
  show tables                                                       1
  select * from image                                               3
```

Here the `select * from image` query was performed three times while tracing.

MySQL: Disk I/O Size Distribution

This matches the `execname` of `mysqld` when block I/O is issued. This will only match disk I/O issued synchronously with `mysqld`; writes buffered by a file system and flushed to disk later will not be matched by this one-liner.

```
server# dtrace -n 'io:::start /execname == "mysqld"/ { @ = quantize(args[0]->b_
bcount); }'
dtrace: description 'io:::start ' matched 6 probes
^C


           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@                                     1
            1024 |@@@@                                     1
            2048 |                                         0
            4096 |                                         0
            8192 |@@@@                                     1
           16384 |                                         0
           32768 |@@@@                                     1
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@                 6
          131072 |                                         0
```

Here we traced several I/O, with most in the 64KB to 128KB range. While tracing, many queries were performed to the MySQL server, yet there were few disk I/Os as a result—either the MySQL cache was returning most of them (as the following scripts can identify) or the disk I/O was asynchronous to MySQL (see Chapter 5, File Systems, for tracing the VFS layer, which enables observing file system I/O while the `mysqld` process is still on-CPU).

Scripts

Table 10-4 summarizes the scripts for MySQL and the providers they use.

mysqld_qsnoop.d

This script traces queries live and prints details including the time and result.

Table 10-4 MySQL Script Summary

Script	Target	Description	Provider
`mysqld_qsnoop.d`	Server	Snoops all queries with info including result	mysql
`mysqld_qchit.d`	Server	Shows query-cache hit rate with missed queries	mysql
`mysqld_qslower.d`	Server	Snoops queries slower than given milliseconds	mysql
`mysqld_pid_qtime.d`	Server	Shows query time distribution plots	pid
`libmysql_snoop.d`	Client	Snoops client queries via libmysqlclient	pid

Script

This script depends on the `query-start` and `query-done` probes firing in the same thread so that the thread-local (`self->`) variables are passed between the action blocks.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-8s %-16s %-18s %5s %3s %s\n", "TIME(ms)", "DATABASE",
            "USER@HOST", "ms", "RET", "QUERY");
        timezero = timestamp;
}

mysql*:::query-start
{
        self->query = copyinstr(arg0);
        self->db = copyinstr(arg2);
        self->who = strjoin(copyinstr(arg3), strjoin("@", copyinstr(arg4)));
        self->start = timestamp;
}

mysql*:::query-done
/self->start/
{
        this->now = (timestamp - timezero) / 1000000;
        this->time = (timestamp - self->start) / 1000000;
        printf("%-8d %-16.16s %-18.18s %5d %3d %s\n", this->now, self->db,
            self->who, this->time, (int)arg0, self->query);
        self->start = 0; self->query = 0; self->db = 0; self->who = 0;
}
```

The output may be shuffled slightly by DTrace: the `TIME(ms)` column can be used for postsorting to see the queries in the correct order.

Examples

The `mysqld_qsnoop.d` script is demonstrated by observing a wiki server using a MySQL database.

> **CLI queries**: The `mysqld_snoop.d` script was used to monitor lookups on a wiki server that utilizes a MySQL database (MediaWiki), while a few queries were tested locally.

```
server# mysqld_qsnoop.d
TIME(ms) DATABASE         USER@HOST           ms RET QUERY
2208     wikidb           wikiuser@localhost   2   0 show tables
5974     wikidb           wikiuser@localhost  63   0 select * from user
8727     wikidb           wikiuser@localhost  22   0 select * from image
```

```
9590     wikidb          wikiuser@localhost     0   0 select * from image
29262    wikidb          wikiuser@localhost     0   1 select * from bogus
^C
```

Note that the first lookup of the image table took 22 ms, followed by 0 ms—most likely because of caching in either the MySQL query cache or the file system cache (the scripts that follow can confirm). The lookup of the bogus table returned 1, error, since this table does not exist.

Production queries: Now `mysqld_snoop.d` was tracing while a page was loaded from this wiki server:

```
server# mysqld_qsnoop.d
TIME(ms) DATABASE         USER@HOST              ms RET QUERY
5110     wikidb           wikiuser@localhost     0   0 BEGIN
5112     wikidb           wikiuser@localhost     0   0 SET /* Database::open  */ sql_m
ode = ''
5115     wikidb           wikiuser@localhost     1   0 SELECT /* Title::getInterwikiLi
nk  */  iw_url,iw_local,iw_trans  FROM `interwiki`  WHERE iw_prefix = 'configuration'
5141     wikidb           wikiuser@localhost     0   0 /* Article::pageData 192.168.1.
132 */ SELECT  page_id,page_namespace,page_title,page_restrictions,page_counter,page_i
s_redirect,page_is_new,page_random,page_touched,page_latest,page_len  FROM `page`  WHE
RE page_namespace = '0' AND page_title = 'Configurat
5145     wikidb           wikiuser@localhost  1684   0 SELECT /* Title::loadRestrictio
ns 192.168.1.132 */  *  FROM `page_restrictions`  WHERE pr_page = '36'
5146     wikidb           wikiuser@localhost     0   0 /* Title::loadRestrictionsFromR
ow 192.168.1.132 */ SELECT  page_restrictions  FROM `page`  WHERE page_id = '36'  LIMI
T 1
5171     wikidb           wikiuser@localhost     0   0 SELECT /* MediaWikiBagOStuff::_
doquery 192.168.1.132 */ value,exptime FROM `objectcache` WHERE keyname='wikidb:pcache
:idhash:36-0!1!0!!en!2'
[...]
```

One of the queries took more than a second, which added noticeable latency to the page load time. This can be investigated further with more DTrace, such as the `mysqld_qslower.d` script.

The actual queries performed are long strings of text, including MediaWiki comments, which in a few cases have been truncated. Increase the `strsize` tunable to avoid the truncation, which can be set either by adding a `#pragma` directive to the script or by using `-x` at the command line. For example, adjust `strsize` to 32 bytes to avoid the text wrapping:

```
server# mysqld_qsnoop.d -x strsize=32
TIME(ms) DATABASE         USER@HOST              ms RET QUERY
8902     wikidb           wikiuser@localhost     0   0 BEGIN
8903     wikidb           wikiuser@localhost     0   0 SET /* Database::open 192.168.1
8905     wikidb           wikiuser@localhost     0   0 /* Article::pageData 192.168.1.
8911     wikidb           wikiuser@localhost     1   0 SELECT /* Title::loadRestrictio
8912     wikidb           wikiuser@localhost     0   0 /* Title::loadRestrictionsFromR
8947     wikidb           wikiuser@localhost     0   0 COMMIT
9249     wikidb           wikiuser@localhost     0   0 BEGIN
```

continues

```
9250    wikidb          wikiuser@localhost     0    0 SET /* Database::open 192.168.1
9253    wikidb          wikiuser@localhost     1    0 SELECT /* LinkBatch::doQuery 19
9258    wikidb          wikiuser@localhost     1    0 SELECT /* MediaWikiBagOStuff::_
[...]
```

mysqld_qchit.d

This shows queries by query-cache hit and miss, which is useful for determining how well the query cache is performing and which queries are missing. This also calculated and printed hit rate while tracing.

Script

Lines 17–25 exist to truncate long queries so that the columns line up. They can be deleted if you want to see the full queries.

```
 1  #!/usr/sbin/dtrace -s
 2
 3  #pragma D option quiet
 4
 5  dtrace:::BEGIN
 6  {
 7          printf("Tracing... Hit Ctrl-C to end.\n");
 8          hits = 0; misses = 0;
 9  }
10
11  mysql*:::query-cache-hit,
12  mysql*:::query-cache-miss
13  {
14          this->query = copyinstr(arg0);
15  }
16
17  mysql*:::query-cache-hit,
18  mysql*:::query-cache-miss
19  /strlen(this->query) > 60/
20  {
21          this->query[57] = '.';
22          this->query[58] = '.';
23          this->query[59] = '.';
24          this->query[60] = 0;
25  }
26
27  mysql*:::query-cache-hit
28  {
29          @cache[this->query, "hit"] = count();
30          hits++;
31  }
32
33  mysql*:::query-cache-miss
34  {
35          @cache[this->query, "miss"] = count();
36          misses++;
37  }
38
39  dtrace:::END
40  {
41          printf("   %-60s %6s %6s\n", "QUERY", "RESULT", "COUNT");
42          printa("   %-60s %6s %@6d\n", @cache);
43          total = hits + misses;
```

```
        printf("\nHits     : %d\n", hits);
        printf("Misses   : %d\n", misses);
        printf("Hit Rate : %d%%\n", total ? (hits * 100) / total : 0);
}
```

Example

To test this script, a table dump of the image table was performed ten times, and a table dump of the user table was performed five times:

```
server# mysqld_qchit.d
Tracing... Hit Ctrl-C to end.
^C
   QUERY                                                    RESULT  COUNT
   select * from user                                         miss      5
   select * from image                                        miss     10

Hits     : 0
Misses   : 15
Hit Rate : 0%
```

All the queries resulted in misses. This was unexpected. After a little investigation of the MySQL configuration, it was discovered that the query cache was not enabled at all! To fix this, the following was added to `/etc/mysql/my.cnf`:

```
query_cache_size= 16M
```

MySQL was then restarted, and this test was repeated:

```
server# mysqld_qchit.d
Tracing... Hit Ctrl-C to end.
^C
   QUERY                                                    RESULT  COUNT
   select * from image                                        miss      1
   select * from user                                         miss      1
   select * from user                                          hit      4
   select * from image                                         hit      9

Hits     : 13
Misses   : 2
Hit Rate : 86%
```

Now hits can be seen for the repeated queries. (Fortunately for this production server, what MySQL did not cache, the file system cache did, which explains the speedup seen in the example of `mysqld_qsnoop.d`.)

mysqld_qslower.d

This traces queries slower than a given value of milliseconds, with details to determine where the latency is.

Script

This script takes the millisecond value as an argument, which is read on line 11 and converted to nanoseconds. If no argument is given, the script still executes with a value of 0 (thanks to the `defaultargs` pragma on line 4), meaning it will trace all requests.

```
 1  #!/usr/sbin/dtrace -s
 2
 3  #pragma D option quiet
 4  #pragma D option defaultargs
 5  #pragma D option switchrate=10hz
 6
 7  dtrace:::BEGIN
 8  {
 9          printf("%5s %5s %5s %5s %s\n", "QRYms", "EXCms", "CPUms",
10              "CACHE", "QUERY");
11          min_ns = $1 * 1000000;
12  }
13
14  mysql*:::query-start
15  {
16          self->query = copyinstr(arg0);
17          self->start = timestamp;
18          self->vstart = vtimestamp;
19  }
20
21  mysql*:::query-cache-hit,
22  mysql*:::query-cache-miss
23  {
24          self->cache = probename == "query-cache-hit" ? "hit" : "miss";
25  }
26
27  mysql*:::query-exec-start
28  {
29          self->estart = timestamp;
30  }
31
32  mysql*:::query-exec-done
33  /self->estart/
34  {
35          self->exec = timestamp - self->estart;
36          self->estart = 0;
37  }
38
39  mysql*:::query-done
40  /self->start && (timestamp - self->start) >= min_ns/
41  {
42          this->time = (timestamp - self->start) / 1000000;
43          this->vtime = (vtimestamp - self->vstart) / 1000000;
44          this->etime = self->exec / 1000000;
45          printf("%5d %5d %5d %5s %s\n", this->time, this->etime, this->vtime,
46              self->cache, self->query);
47  }
48
```

```
mysql*:::query-done
{
        self->start = 0; self->vstart = 0; self->exec = 0;
        self->cache = 0; self->query = 0;
}
```

The columns printed are as follows:

QRYms: Query time, milliseconds

EXCms: Execution time, milliseconds

CPUms: On-CPU time, milliseconds

Example

Here the script was used to trace any query that took one millisecond or longer:

```
server# mysqld_qslower.d 1
QRYms EXCms CPUms CACHE QUERY
    2     1     2  miss show tables
   25    24     5  miss select * from pagelinks
    4     4     4  miss select * from pagelinks
```

The `show tables` query showed 2 milliseconds, which was entirely on-CPU time (code path), with 1 millisecond in the execution stage.

The next query (`select * from pagelinks`) took 25 milliseconds, 24 of which were in the execution stage. However, only 5 milliseconds were on-CPU, meaning that most of this time was waiting off-CPU, most probably on disk I/O to satisfy the query.

This query was repeated, the second time taking 4 milliseconds, all on-CPU. Here the query-cache was disabled, and although the file system appears to be returning the data from its cache (no off-CPU time waiting on disk I/O), there was still significant latency as the (file system–cached) database files were reread for the query. Here's an example of enabling the cache and running the script with no arguments to trace all events:

```
server# mysqld_qslower.d
QRYms EXCms CPUms CACHE QUERY
    0     0     0   hit select  from pagelinks
```

This shows the same query is returning in 0 ms (rounded to zero), instead of 4 ms. This makes a case for using the query cache over the file system cache.

mysqld_pid_qtime.d

This script traces queries and shows their time as distribution plots by query string. It was written to demonstrate using the pid provider and as such is expected to work only on a particular version of the MySQL source (5.1).

Script

This script traces MySQL internal functions and arguments (the `dispatch_command()` function, with `arg2`) for this MySQL version. This can be rewritten to match different MySQL versions and internals or to use the mysql provider if available.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

pid$target::*dispatch_command*:entry
{
        self->query = copyinstr(arg2);
        self->start = timestamp;
}

pid$target::*dispatch_command*:return
/self->start/
{
        @time[self->query] = quantize(timestamp - self->start);
        self->query = 0; self->start = 0;
}

dtrace:::END
{
        printf("MySQL query execution latency (ns):\n");
        printa(@time);
}
```

Example

In the following output, the `select * from months` query had one execution take between 8 and 16 milliseconds, 11 executions take between 0.5 and 1.0 milliseconds, and 9 executions take between 131 and 262 microseconds. Presumably, the slow execution (8 ms to 16 ms) was the first query, which put the data in memory (which `mysqld_qslower.d`, or a pid-based variant, could confirm).

```
server# mysqld_pid_qtime.d -p `pgrep -x mysqld`
Tracing... Hit Ctrl-C to end.
^C
MySQL query execution latency (ns):
```

```
show tables
        value  ------------- Distribution ------------- count
       131072 |                                         0
       262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
       524288 |                                         0

select * from months
        value  ------------- Distribution ------------- count
        65536 |                                         0
       131072 |@@@@@@@@@@@@@@@@@                        9
       262144 |                                         0
       524288 |@@@@@@@@@@@@@@@@@@@@@                    11
      1048576 |                                         0
      2097152 |                                         0
      4194304 |                                         0
      8388608 |@@                                       1
     16777216 |                                         0

select * from words where name < 'fish'
        value  ------------- Distribution ------------- count
      8388608 |                                         0
     16777216 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    11
     33554432 |@@@                                      1
     67108864 |                                         0
```

The process ID was provided to the script using the `-p` option and running the Oracle Solaris `pgrep(1)` utility.

libmysql_snoop.d

This script traces queries on the client side for software using the MySQL C client library (libclientmysql) and shows the queries performed along with time and result.

Script

This uses the pid provider to trace the internals of the libmysqlclient library. The pid provider is usually considered an unstable interface, however; here it is used to trace the MySQL C API—a documented[3] interface that is unlikely to change quickly.

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-8s %6s %3s %s\n", "TIME(ms)", "Q(ms)", "RET", "QUERY");
```

continues

3. See *http://dev.mysql.com/doc/refman/5.1/en/c.html* for MySQL 5.1.

```
        timezero = timestamp;
}

pid$target::mysql_query:entry,
pid$target::mysql_real_query:entry
{
        self->query = copyinstr(arg1);
        self->start = timestamp;
}

pid$target::mysql_query:return,
pid$target::mysql_real_query:return
/self->start/
{
        this->time = (timestamp - self->start) / 1000000;
        this->now = (timestamp - timezero) / 1000000;
        printf("%-8d %6d %3d %s\n", this->now, this->time, arg1, self->query);
        self->start = 0; self->query = 0;
}
```

Example

Here the `mysqld_client.d` script was used to trace queries made by a PHP program that was performing wiki maintenance. The `strsize` tunable was used to truncate output, as shown earlier with `mysqld_qsnoop.d`:

```
client# mysql_client.d -x strsize=60 -c 'php rebuildrecentchanges.php'
TIME(ms)  Q(ms) RET QUERY
6433          0   0 SET /* Database::open  */ sql_mode = ''
6454         20   0 DELETE /* Database::delete  */ FROM `recentchanges`
6457          1   0 INSERT /* rebuildRecentChangesTablePass1  */  INTO `recentc
6458          0   0 SELECT /* -2  */ rc_cur_id,rc_this_oldid,rc_timestamp FROM
6469         10   0 SELECT /* -2  */ DISTINCT rc_user FROM `recentchanges` LEFT
6469          0   0 SELECT /* -2  */ DISTINCT rc_user FROM `recentchanges` LEFT
```

Each query performed is traced from the client, including the query time (`Q(ms)`) and return value (`RET`). The longest query performed was a `DELETE` from `recentchanges`, taking 20 milliseconds.

The output did include the text output from the PHP script, which has been removed from this example to focus on the DTrace output.

See Also

Much more is possible with DTrace and MySQL; for more information, see the following references:

"Tracing mysqld using DTrace" in the MySQL Reference Manual[4]

4. *http://dev.mysql.com/doc/refman/5.5/en/dba-dtrace-server.html for MySQL 5.5*

"MySQL and DTrace" (January 2009) by MC Brown, at MySQL University[5]

MySQL `top` using DTrace[6]

Chapter 9 (for both the server and client)

The MySQL Reference Manual has subsections for the probe groups, many of which have examples of DTrace code. For example, the following is the example output for the first sample script in the "Statement Probes" section:[7]

```
Query                                           RowsU    RowsM    Dur (ms)
select * from t2                                0        275      0
insert into t2 (select * from t2)               0        275      9
update t2 set i=5 where i > 75                  110      110      8
update t2 set i=5 where i < 25                  254      134      12
delete from t2 where i < 5                      0        0        0
```

Although the DTrace script is not included here, the output is enough to demonstrate further capabilities: It includes rows updated, rows matched, and duration by query.

PostgreSQL

A DTrace provider for PostgreSQL[8] was developed and added to the server source, with basic probes appearing in version 8.2 and extended probes appearing in 8.4. The provider is available only when PostgreSQL has been compiled with the `--enable-dtrace` option. Probes are listed in the PostgreSQL manual in the "Dynamic Tracing" chapter,[9] and version 8.4 probes are demonstrated here. Table 10-5 is a partial listing of the available probes.

A few examples of using DTrace on PostgreSQL are shown here as one-liners and scripts to retrieve query strings and times. These scripts can be enhanced to include whatever other information is of interest from the operating system so that it can be examined in the context of PostgreSQL queries.

5. *http://forge.mysql.com/wiki/Using_DTrace_with_MySQL*
6. *http://milek.blogspot.com/2010/01/mysql-top.html*
7. *http://dev.mysql.com/doc/refman/5.5/en/dba-dtrace-ref-statement.html*
8. *www.postgresql.org*
9. See *www.postgresql.org/docs/8.4/static/dynamic-trace.html* for PostgreSQL 8.4.

Table 10-5 PostgreSQL DTrace Probes

Probe name	Description
`transaction-start`	Probe that fires at the start of a new transaction. `arg0` is the transaction ID.
`transaction-commit`	Probe that fires when a transaction completes successfully. `arg0` is the transaction ID.
`query-start`	Probe that fires when the processing of a query is started. `arg0` is the query string.
`query-done`	Probe that fires when the processing of a query is complete. `arg0` is the query string.
`checkpoint-start`	Probe that fires when a checkpoint is started. `arg0` holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate, or force.
`checkpoint-done`	Probe that fires when a checkpoint is complete. (The probes listed next fire in sequence during checkpoint processing.) `arg0` is the number of buffers written. `arg1` is the total number of buffers. `arg2`, `arg3`, and `arg4` contain the number of `xlog` file(s) added, removed, and recycled, respectively.
`buffer-sync-start`	Probe that fires when we begin to write dirty buffers during checkpoint (after identifying which buffers must be written). `arg0` is the total number of buffers. `arg1` is the number that are currently dirty and need to be written.
`buffer-sync-written`	Probe that fires after each buffer is written during checkpoint. `arg0` is the ID number of the buffer.
`buffer-read-start`	Probe that fires when a buffer read is started. `arg0` and `arg1` contain the fork and block numbers of the page (but `arg1` will be -1 if this is a relation extension request). `arg2`, `arg3`, and `arg4` contain the tablespace, database, and relation OIDs identifying the relation. `arg5` is true for a local buffer, false for a shared buffer. `arg6` is true for a relation extension request, false for normal read.
`buffer-read-done`	Probe that fires when a buffer read is complete. `arg0` and `arg1` contain the fork and block numbers of the page (if this is a relation extension request, `arg1` now contains the block number of the newly added block). `arg2`, `arg3`, and `arg4` contain the tablespace, database, and relation OIDs identifying the relation. `arg5` is true for a local buffer, false for a shared buffer. `arg6` is true for a relation extension request, false for normal read. `arg7` is true if the buffer was found in the pool, false if not.

One-Liners

Here are some PostgreSQL DTrace one-liners.

postgresql Provider

These one-liners are for tracing events on the PostgreSQL server.

PostgreSQL: query trace by query string:

```
dtrace -n 'postgresql*:::query-start { trace(copyinstr(arg0)) }'
```

PostgreSQL: query count summary by query string:

```
dtrace -n 'postgresql*:::query-start { @[copyinstr(arg0)] = count(); }'
```

PostgreSQL: count query events:

```
dtrace -n 'postgresql*:::query-* { @[probename] = count(); }'
```

PostgreSQL: count buffer read/flush events:

```
dtrace -n 'postgresql*:::buffer-* { @[probename] = count(); }'
```

PostgreSQL: count lock events:

```
dtrace -n 'postgresql*:::lwlock-*,postgresql*:::lock-* { @[probename] = count(); }'
```

PostgreSQL: checkpoint trace with type integer:

```
dtrace -n 'postgresql*:::checkpoint-start { printf("PID %d, type %d", pid, arg0); }'
```

PostgreSQL: server query status trace (simple snoop):

```
dtrace -qn 'postgresql*:::statement-status { printf("%d ns, PID %d, %s\n",
timestamp, pid, copyinstr(arg0)); }'
```

pid Provider

The following are examples of tracing the PostgreSQL 8.2 server internal functions using the pid provider; these are likely to need the function name and arguments adjusted for other versions.

PostgreSQL server: trace queries:

```
dtrace -qn 'pid$target::exec_simple_query:entry { printf("%Y   %s\n",
walltimestamp, copyinstr(arg0)); }' -p PID
```

PostgreSQL server: count queries:

```
dtrace -n 'pid$target::exec_simple_query:entry { @[copyinstr(arg0)] =
count(); }' -p PID
```

Apart from `exec_simple_query()`, the `pg_parse_query()` function may exist that can serve the same purpose: retrieving the query string.

One-Liner Selected Examples

Here are some selected examples.

PostgreSQL: Server Query Status Trace (Simple Snoop)

This one-liner is a simple (and rough) way to snoop PostgreSQL server activity, by tracing changes to its status string:

```
# dtrace -qn 'postgresql*:::statement-status { printf("%d ns, PID %d, %s\n", timestamp
, pid, copyinstr(arg0)); }'
2974757108112944 ns, PID 218225, select * from images;
2974757108326123 ns, PID 218225, <IDLE>
2974772955404114 ns, PID 218225, copy words from '/usr/dict/words';
2974774714391125 ns, PID 218225, <IDLE>
2974796403248508 ns, PID 218254, autovacuum: ANALYZE public.words
```

The current server time is printed as nanoseconds, allowing time during states to be calculated as the delta between lines.

Scripts

Table 10-6 summarizes the following scripts for PostgreSQL and the providers they use.

Table 10-6 PostgreSQL Script Summary

Script	Target	Description	Provider
pg_qslower.d	Server	Snoops queries slower than given milliseconds	postgresql
pg_pid_qtime.d	Server	Shows query time distribution plots by query string	pid

pg_qslower.d

This traces queries slower than a given value of milliseconds, with details to determine where the latency is. If the provided value is 0, this script traces all queries.

Script

This script is based on `mysql_qslower.d`. The millisecond argument is processed on line 11 and is zero by default thanks to line 4.

```
1      #!/usr/sbin/dtrace -s
2
3      #pragma D option quiet
4      #pragma D option defaultargs
5      #pragma D option switchrate=10hz
6
7      dtrace:::BEGIN
8      {
9              printf("%-8s %5s %5s %5s %s\n", "TIMEms", "QRYms", "EXCms", "CPUms",
10               "QUERY");
11           min_ns = $1 * 1000000;
12           timezero = timestamp;
13     }
14
15     postgresql*:::query-start
16     {
17           self->start = timestamp;
18           self->vstart = vtimestamp;
19     }
20
21     postgresql*:::query-execute-start
22     {
23           self->estart = timestamp;
24     }
25
26     postgresql*:::query-execute-done
27     /self->estart/
28     {
29           self->exec = timestamp - self->estart;
30           self->estart = 0;
31     }
32
33     postgresql*:::query-done
34     /self->start && (timestamp - self->start) >= min_ns/
35     {
36             this->now = (timestamp - timezero) / 1000000;
```

continues

```
                this->time = (timestamp - self->start) / 1000000;
                this->vtime = (vtimestamp - self->vstart) / 1000000;
                this->etime = self->exec / 1000000;
                printf("%-8d %5d %5d %5d %s\n", this->now, this->time, this->etime,
                  this->vtime, copyinstr(arg0));
        }

        postgresql*:::query-done
        {
                self->start = 0; self->vstart = 0; self->exec = 0;
        }
```

The columns printed are as follows:

TIMEms: Elapsed time since tracing started, milliseconds

QRYms: Query time, milliseconds

EXCms: Execution time, milliseconds

CPUms: Time on-CPU, milliseconds

Example

Here's an example of tracing queries taking ten milliseconds or longer:

```
server# pg_qslower.d 10
TIMEms  QRYms EXCms CPUms QUERY
1031     2201  2191    86 CREATE DATABASE wikidb;
10229      71    28     4 create table images ( name varchar(80), index int );
15872      32     0     1 INSERT INTO images VALUES ( 'fred', 1 );
23735      22     0     1 create table words ( word varchar(80) );
28231    2087  2075    85 copy words from '/usr/dict/words';
^C
```

There were two queries that took longer than two seconds: a `CREATE DATABASE` query and a `copy` query. For both of these, only about 80 ms was spent on-CPU, suggesting that the rest of the time was likely spent waiting on disk I/O.

pg_pid_qtime.d

This script traces queries and shows their time as distribution plots by query string. It was written to demonstrate using the pid provider and as such may work only on a particular version of the PostgreSQL source (8.2).

Script

This script traces internal functions and arguments (the `exec_simple_query()` function, with `arg0`) for this PostgreSQL version. This can be rewritten to match different PostgreSQL versions and internals or to use the postgres provider if available.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

pid$target::exec_simple_query:entry
{
        self->query = copyinstr(arg0);
        self->start = timestamp;
}

pid$target::exec_simple_query:return
/self->start/
{
        @time[self->query] = quantize(timestamp - self->start);
        self->start = 0; self->query = 0;
}

dtrace:::END
{
        printf("PostgreSQL simple query execution latency (ns):\n");
        printa(@time);
}
```

Example

Here the `pg_pid_qtime.d` script was aimed at a PostgreSQL server process (PID 86008, `postmaster`) that was serving an command-line instance of `psql`. While tracing, a table was populated and searched:

```
server# pg_pid_qtime.d -p 86008
Tracing... Hit Ctrl-C to end.
^C
PostgreSQL simple query execution latency (ns):

  select * from images;
           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
          262144 |                                         0

  select * from words;
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@@@                                     1
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@                 6
        67108864 |@@@@@@@@@@@@                             3
       134217728 |                                         0

  copy words from '/usr/dict/words';
           value  ------------- Distribution ------------- count
       268435456 |                                         0
       536870912 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
      1073741824 |                                         0
```

The slowest query was a "copy" from the 25,143-line file `/usr/dict/words` to populate the words table, which took more than 500 ms. Several `select * from words` queries were then executed, which mostly fell in the 33-ms to 67-ms range.

See Also

Much more is possible with DTrace and PostgreSQL; for more information, see the following references:

26.4 "Dynamic Tracing" in the PostgreSQL documentation[10]

The PostgreSQL-DTrace-Toolkit[11] by Robert Lor

Chapter 9 (for both the server and client)

Oracle

There is not currently an Oracle DTrace provider built in to the database software; until there is, it can be observed like any other application, by examining how it uses system resources such as CPUs, disks, and the network. See the following chapters of this book:

Chapter 3, System View

Chapter 4, Disk I/O (especially if raw devices are used)

Chapter 5, File Systems

Chapter 6, Network Lower-Level Protocols

Chapter 9, Applications (for both the server and client)

Although the database internals can be observed using the pid provider or by gathering user stack traces in various probes, it is generally not a useful exercise on the Oracle Database, unless working a case directly with Oracle Support and someone with knowledge of and access to the source code.

Examples

This example is taken from a 128 CPU SPARC system running Solaris 10, executing a CPU-intensive decision support query.

10. See *www.postgresql.org/docs/8.4/static/dynamic-trace.html*.

11. This is currently at *http://pgfoundry.org/projects/dtrace/*.

We start with a system view:

```
solaris# vmstat 1 10
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s1 s2 s3 s4   in   sy   cs us sy id
 0 0 0 96761632 88160736 15 25 28 0 0 0  3  0  4 -0  4 5025 1446 1212  2  0 98
 59 0 0 68166200 59685864 166 182 0 1 1 0 0 0  0  0  0 15085 65474 4853 95 5 0
 59 0 0 68164080 59690376 0 0 0 0  0  0  0  0  0  0 15257 85574 4621 94 6 0
 66 0 0 68164056 59690352 0 0 0 0  0  0  0  0  0  0 14583 46688 4292 96 4 0
 60 0 0 68162992 59689280 0 0 0 0  0  0  0  0  0  0 15371 82698 4838 94 6 0
 57 0 0 68158456 59684736 0 0 0 0  0  0  0  0  0  0 14815 56748 4446 95 5 0
 58 0 0 68149888 59670304 59 107 0 2 2 0 0  0  0  0  0 15343 87177 4766 94 6 0
 67 0 0 68144808 59658880 395 395 0 0 0 0 0 0  0  0  0 14841 68232 4494 95 5 0
 58 0 0 68138328 59651960 23 310 0 2 2 0 0  0  0  0  0 15403 88676 4955 93 7 0
 65 0 0 68138248 59651424 20 146 0 0 0 0 0  0  0  0  0 14694 59987 4309 95 5 0
```

We can see from the `vmstat` data the CPUs are very busy, virtually all in user mode, and the run queue depth is running at about 60 runnable threads in the queues. Note that number does not include threads running on a CPU. This is a 128 CPU system, so we have 128 running threads, plus 60 or so waiting to run. Other data examined (not shown here) indicated there was minimal disk I/O, indicating the data for the query was being satisfied out of the memory cache (Oracle SGA db_block_buffers) and virtually no network I/O.

Given the CPUs are very busy and the run queues are consistently nonzero, we should take a look at run queue latency. First we'll do this with `prstat(1)` with the `m` flag to monitor thread microstates:

```
solaris# prstat -mc 5 2
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
  2298 dbbench   46 2.5 0.1 0.0 0.0 0.0  50 1.2   2 320  3K   0 oracle.darre/2
  2044 dbbench   45 2.4 0.1 0.0 0.0 0.0  50 2.7   2 312  3K   0 oracle.darre/2
  2108 dbbench   45 2.2 0.1 0.0 0.0 0.0  50 3.1   3 308  3K   0 oracle.darre/2
  2210 dbbench   45 2.3 0.1 0.0 0.0 0.0  50 3.1   4 305  2K   0 oracle.darre/2
[...]
  2122 dbbench   25 1.5 0.1 0.0 0.0 0.0  51  23   4 173  2K   0 oracle.darre/2
  2270 dbbench   25 1.1 0.1 0.0 0.0 0.0  50  24   0 174  1K   0 oracle.darre/2
  2232 dbbench   26 0.8 0.1 0.0 0.0 0.0  50  23   3 181  1K   0 oracle.darre/2
  2052 dbbench   25 1.3 0.1 0.0 0.0 0.0  50  23   6 170  1K   0 oracle.darre/2
  2062 dbbench   25 1.2 0.1 0.0 0.0 0.0  50  24   2 170  1K   0 oracle.darre/2
Total: 355 processes, 752 lwps, load averages: 182.06, 132.24, 104.07
```

We can see the LAT values (run queue latency) are very high for many of the processes (more than are shown here), getting more than 20 percent. With DTrace, we can measure the run queue latency and track run queue depth both per-CPU and systemwide. For Solaris:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      printf("Sampling at 1001 Hertz... Hit Ctrl-C to end.\n");
}

profile:::profile-1001hz
{
      @["Per-CPU disp queue length:"] =
          lquantize(curthread->t_cpu->cpu_disp->disp_nrunnable, 0, 64, 1);
}

Script cpudispqlen.d
```

Now we run the `cpudispqlen.d` script on this server:

```
solaris# ./cpudispqlen.d
Sampling at 1001 Hertz... Hit Ctrl-C to end.

  Per-CPU disp queue length:
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@                 283778
               1 |@@@@@@@@@@@@@                            161042
               2 |@@@                                      34316
               3 |                                         3197
               4 |                                         1
               5 |                                         0
```

The per-CPU run queue depth ranges show zero to three threads in the queues, with a peak for one sample of four to five threads in the queues. Note again these are the per-CPU run queues.

Looking at run queue depth systemwide yields the following:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
      printf("Sampling at 1001 Hertz... Hit Ctrl-C to end.\n");
}

profile:::profile-1001hz
{
      @["System wide disp queue length:"] =
          sum(curthread->t_cpu->cpu_disp->disp_nrunnable);
}

profile:::tick-1sec
{
```

```
      normalize(@, 1001);
      printa(@);
      trunc(@);
}
Script sysdispqlen.d
```

Running `sysdispqlen.d` yields the following:

```
solaris# ./sysdispqlen.d
Sampling at 1001 Hertz... Hit Ctrl-C to end.
  System wide disp queue length:                                  46
  System wide disp queue length:                                  50
  System wide disp queue length:                                  49
  System wide disp queue length:                                  33
  System wide disp queue length:                                  31
  System wide disp queue length:                                  31
  System wide disp queue length:                                  42
  System wide disp queue length:                                  52
  System wide disp queue length:                                  61
  System wide disp queue length:                                  56
^C
```

Note that the values here are consistent with what was reported by `vmstat` (the `r` column). Also, these values represent the number of threads systemwide sitting on run queues waiting to run—the number of running threads is not included. We know from the CPU utilization data all the CPUs are busy running threads, so for this system, we have 128 + 50 (178) active threads. When the number of active threads exceeds the number of CPUs, we will experience some run queue latency. We can use DTrace to measure the time threads spend waiting in run queues.

```
#!/usr/sbin/dtrace -s

sched:::enqueue
{
      s[args[0]->pr_lwpid, args[1]->pr_pid] = timestamp;
}

sched:::dequeue
/this->start = s[args[0]->pr_lwpid, args[1]->pr_pid]/
{
      @[args[2]->cpu_id] = quantize(timestamp - this->start);
      s[args[0]->pr_lwpid, args[1]->pr_pid] = 0;
}

Script qtime.d
```

The `qtime.d` script tracks the time threads spend on a run queue on a per-CPU basis.

```
solaris# ./qtime.d

        1
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@                     90
           32768 |@@@                                      15
           65536 |@@@                                      12
          131072 |                                         2
          262144 |@@                                       7
          524288 |@                                        3
         1048576 |@                                        6
         2097152 |@                                        5
         4194304 |@@@                                      13
         8388608 |@@@@@                                    21
        16777216 |@@                                       10
        33554432 |                                         0

       60
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@                    97
           32768 |@@@@                                     18
           65536 |                                         2
          131072 |                                         0
          262144 |                                         1
          524288 |@@                                       9
         1048576 |@                                        3
         2097152 |@@                                       11
         4194304 |@@@@                                     20
         8388608 |@@@                                      13
        16777216 |@@@                                      12
        33554432 |                                         0
[...]
       99
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@                       90
           32768 |@@                                       9
           65536 |@@@@                                     19
          131072 |                                         1
          262144 |                                         0
          524288 |@@                                       10
         1048576 |@@                                       8
         2097152 |@                                        5
         4194304 |@@                                       10
         8388608 |@@@@@@                                   30
        16777216 |@@@                                      17
        33554432 |                                         2
        67108864 |                                         0

      100
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@                       85
           32768 |@@@@                                     19
           65536 |@                                        6
          131072 |                                         1
          262144 |                                         0
          524288 |@                                        6
         1048576 |@                                        3
         2097152 |@@                                       9
         4194304 |@@                                       10
         8388608 |@@@@@@                                   28
        16777216 |@@@@@                                    22
        33554432 |                                         0
```

The data has once again been truncated for space purposes. For each CPU, a quantize aggregation is generated, showing the time values in nanoseconds on the left column and the number of occurrences of threads waiting on a run queue for that time period on the right. The data shows that typical run latency is 16 us to 32 us, with a grouping of outliers in the 8-ms to 32-ms range. The numbers suggest that roughly 30 percent of the threads fall within the 8-ms to 32-ms range.

Unfortunately, the overall effect the 30 percent worse-case run queue latency is having on overall query time is difficult to measure in any detail, because of the complexity of assessing the benefits of concurrency vs. added run queue latency. For example, the Oracle parameters for this system could be altered to reduce the number of query slaves spawned to run this query, to say no more than 128 (the number of CPUs). That would certainly improve run queue latency but would also potentially impact the total time for the query to complete as a result of the reduced number of slaves and the additional work needed to be done by the query slave processes.

On the same system, a different DSS query was executed with a similar profile in terms of system utilization. For this query, we observed the disk I/O component, starting with running `iostat(1M)`:

```
solaris# iostat -xnz 5 10
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...]
   57.3    0.0 56616.3    0.0  0.0  0.4    0.0    6.4   0  29
c18t21000024FF206BB9d167
   11.5    0.0 11446.9    0.0  0.0  0.1    0.0    5.9   0   6 c7t21000024FF206CFDd164
   71.3    0.0 71040.8    0.0  0.0  0.4    0.0    5.9   0  32 c7t21000024FF206CFDd163
   47.4    0.0 47563.2    0.0  0.0  0.3    0.0    6.4   0  24
c18t21000024FF206BB9d164
   57.2    0.0 56753.0    0.0  0.0  0.3    0.0    5.7   0  27 c7t21000024FF206CFDd162
   39.3    0.0 39221.1    0.0  0.0  0.3    0.0    6.6   0  21
c18t21000024FF206BB9d163
   77.4    0.0 77268.7    0.0  0.0  0.4    0.0    5.8   0  33 c7t21000024FF206CFDd161
   42.5    0.0 42602.8    0.0  0.0  0.3    0.0    6.5   0  22
c18t21000024FF206BB9d162
   46.6    0.0 46516.6    0.0  0.0  0.3    0.0    7.0   0  23
c18t21000024FF206BB9d161
   99.8    0.0 99457.8    0.0  0.0  0.6    0.0    5.8   0  42 c7t21000024FF206CFDd159
0.0 40755.8    0.0  0.0  0.3    0.0    6.5   0  22 c18t21000024FF206BB9d160
[…]
```

Again, we're showing truncated output here for space purposes. We can see several disks sustaining a moderate level of reads-per-second (r/sec), throughput ranging from 11MB/sec to 99MB/sec (kr/sec), and about 6 milliseconds of latency (`asvc_t`).

We can get a more detailed view of disk I/O latency using DTrace. We first applied methods described in Chapter 4, Disk I/O, to measure latency. However, we found the `io:::done` probe was not firing because of the use of asynchronous I/O

APIs used by the software for disk reads and writes. This was likely because of a bug in the io provider on this version of Solaris. As a workaround, we determined which disk device driver was being used (ssd) and measured latency using the unstable fbt provider. First, we observed which ssd driver routines were being called:

```
solaris# dtrace -n 'fbt:ssd::entry { @[probefunc] = count(); } tick-5s { exit(0); }'
dtrace: description 'fbt:ssd::entry ' matched 292 probes

  ssdopen                                                             11
  ssd_buf_iodone                                                    3345
  ssd_ddi_xbuf_done                                                 3345
  ssd_ddi_xbuf_get                                                  3345
  ssd_destroypkt_for_buf                                            3345
  ssd_mapblockaddr_iodone                                           3345
  ssd_return_command                                                3345
  ssd_xbuf_dispatch                                                 3345
  ssdintr                                                           3345
  ssd_fill_scsi1_lun                                                3346
  ssd_setup_rw_pkt                                                  3346
  ssd_add_buf_to_waitq                                              3347
  ssd_core_iostart                                                  3347
  ssd_ddi_xbuf_qstrategy                                            3347
  ssd_initpkt_for_buf                                               3347
  ssd_mapblockaddr_iostart                                          3347
  ssd_xbuf_init                                                     3347
  ssd_xbuf_iostart                                                  3347
  ssd_xbuf_strategy                                                 3347
  ssdaread                                                          3347
  ssdinfo                                                           3347
  ssdmin                                                            3347
  ssdstrategy                                                       3347
  ssd_start_cmds                                                    6692
```

Using the `ssdstrategy` routine as the entry point and `ssd_buf_iodone` as the I/O completed point, we developed `ssdlatency.d`:

```
#!/usr/sbin/dtrace -s

fbt:ssd:ssdstrategy:entry
{
      start[arg0] = timestamp;
}

fbt:ssd:ssd_buf_iodone:entry
/start[arg2]/
{
      @time["ssd I/O latency (ns)"] = quantize(timestamp - start[arg2]);
 start[arg2] = 0;
}

Script ssdlatency.d

solaris# ./ssdlatency.d

```

```
ssd I/O latency (ns)
          value  ------------- Distribution ------------- count
         131072 |                                         0
         262144 |                                         3
         524288 |                                         47
        1048576 |                                         32
        2097152 |                                         58
        4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  4560
        8388608 |                                         22
       16777216 |                                         1
       33554432 |                                         0
```

The quantize aggregation shows the disk latency in the 4-ms to 8-ms range, which is consistent with the iostat data. However, what gets lost in the averaging of tools like iostat is outliers. The quantize aggregation shows we did have an outlier in the 16-ms to 32-ms range. In this specific example, it was only one occurrence (and 22 I/Os in the 8-ms to 16-ms range), so it is not a cause for concern, but the key point here is to be aware that stat utilities tend to generate averages, which can hide periodic events that fall well outside the average range. Using DTrace, we can reveal these outliers.

Overall, looking at the two queries, we found there was dispatcher queue latency because of the large number of runnable, compute-bound threads, and we measured the latency accurately using DTrace. We also confirmed that the disks were performing as expected.

Summary

In this chapter, we introduced DTrace scripts and one-liners that utilize the database-specific providers. We also showed using DTrace with Oracle, which as of this writing does not have a DTrace provider but can still be observed and analyzed with DTrace. We showed, with DTrace, database operation on both the server and the client can be examined in detail, identifying common queries and clients, query time broken down by query stage, query cache performance, and other internal behavior of the database. By having database context in DTrace, other system events such as disk and network I/O can be correlated to queries, as well as CPU time consumed. Although the Oracle Database currently does not have a provider to provide query context, its operation was examined as an application, including its usage of the CPUs and disks.

This page intentionally left blank

11

Security

Since DTrace can examine custom events on the system with whatever additional data is of interest, it can be applied for various uses in computer security. These include the following:

- Sniffing, such as real-time forensics
- Monitoring:
  - Custom auditing
  - Host-based Intrusion Detection Systems (HIDS)
- Policy enforcement
- Security debugging:
  - Privilege debugging
  - Reverse engineering

Scripts are provided in this chapter to demonstrate these uses. These and additional topics including DTrace privileges and DTrace-based attacks are discussed first.

Privileges, Detection, and Debugging

In this section, we discuss the Solaris privileges associated with using DTrace and how DTrace can be used in several important security scenarios.

DTrace Privileges

By default, only the root user (administrator) can use DTrace. Other users see the following:

```
$ /usr/sbin/dtrace -n 'BEGIN'
dtrace: failed to initialize dtrace: DTrace requires additional privileges
```

Oracle Solaris has a privileges facility (see `privileges(5)`) that allows specific authorizations to be given to processes using the `ppriv(1)` command. They can also be assigned to user accounts using `usermod(1M)`, which saves the privileges to `/etc/user_attr` (companion of `/etc/passwd` and `/etc/shadow`) so that they are granted to the user's login shell process. The available DTrace privileges are summarized in Table 11-1 and are explained in greater detail in the DTrace Guide.[1]

For Oracle Solaris zones, these privileges need to be explicitly granted to nonglobal zones for the zone administrators (root) to be allowed to access DTrace providers. This is performed using the `zonecfg(1M)` command to add these DTrace privileges to the `limitpriv` property. For example, the following command adds the `dtrace_proc` and `dtrace_user` privileges to the nonglobal zone named `zone01`:

```
solaris# zonecfg -z zone01 set limitpriv=default,dtrace_proc,dtrace_user
```

The zone must be rebooted for this to take effect.

Table 11-1 Oracle Solaris Privileges for DTrace

Privilege	Description
`dtrace_proc`	Allows use of the pid and fasttrap providers. User may affect performance of their own processes with DTrace enablings.
`dtrace_user`	Allows use of the syscall and profile providers to inspect the user's own processes. User may affect the performance of other users' processes with DTrace enablings.
`dtrace_kernel`	Allow all providers with the exception of pid and fasttrap providers and all actions with the exception of destructive actions.

1. *http://wikis.sun.com/display/DTrace/Security*

DTrace-Based Attacks

DTrace can examine all events on a system including private data from running applications and can also modify user-land data (the `copyout()` destructive action) as well as run arbitrary commands (`system()`). Before becoming concerned about DTrace-based attacks, we should stress that only privileged users can use DTrace—usually the "root" user, as explained in the previous section. These privileged users can already perform such malicious acts using other system tools.

For example, the "Scripts" section includes `sshkeysnoop.d`, which shows SSH passwords as they are typed. Similar functionality can be performed with existing debuggers; here's an example on Solaris:

```
# ps -ef | grep ssh
    root 129318 129270   0 08:10:52 pts/2       0:00 ssh brendan@mars
[...]
# truss -p 129318
read(6, 0x0804740F, 1)          (sleeping...)
read(6, " s", 1)                                = 1
read(6, " e", 1)                                = 1
read(6, " c", 1)                                = 1
read(6, " r", 1)                                = 1
read(6, " e", 1)                                = 1
read(6, " t", 1)                                = 1
read(6, " 1", 1)                                = 1
read(6, " 2", 1)                                = 1
read(6, " 3", 1)                                = 1
read(6, "\n", 1)                                = 1
[...]
```

Here the password secret123 was sniffed one keystroke at a time using `truss(1)`, a standard Solaris tool that has existed for more than 20 years. Although DTrace can do this more easily as shown by `sshkeysnoop.d`, it has not newly introduced the technical capability to do this.

Despite the existence of tools such as `dtrace(1M)` and `truss(1)`, the operating system is still secure from attacks based on these tools, since they cannot be executed without administrator privileges. Put differently, if an attacker can execute these tools, they have already broken into the system.

Sniffing

Since DTrace can examine any data from the operating system's user or kernel address space, it can be used to examine any user session data on the system, including plain text from applications before encryption is performed. In the security context, this is known as *sniffing*, and because of its capabilities, DTrace is the ultimate sniffer.

The sniffing scripts shown later in this chapter demonstrate this ability as an academic exercise—they are not intended for real-world usage. Since they can examine user-land data including keystrokes and passwords (as can other tools; see DTrace-based attacks), consider privacy concerns (including laws) before using them.

A real-world use for sniffing capabilities may exist, such as to perform real-time forensics[2] by capturing data on an attack in progress (see `cuckoo.d` in the "Scripts" section).

Security Audit Logs

Although DTrace offers incredible visibility into a system, it is designed to be a debugging and analysis tool, not a monitoring or logging tool. It's important to consider that DTrace will drop events when under pressure and can abort executing altogether.

The possibility of dropping events can make DTrace unsuitable for generating security audit logs, which are required to be reliable, authentic, and complete (non-repudiation). An attacker may be able to generate sufficient load to cause DTrace to either miss events or abort entirely. Although the likelihood of DTrace dropping events can be minimized by adjusting tunables (increasing buffer sizes and switch rate and using the destructive pragma), it cannot be eliminated. The best form of security audit logs are those designed for the purpose, such as Oracle Solaris Auditing logs.

This doesn't mean that there are no uses for DTrace in security logging; a log used for intrusion detection may still identify intrusion events, even if the log has become incomplete because of dropped events. DTrace also has much finer-grained capabilities than Solaris Auditing, such as using predicates to match events on certain file and directory names, groups of users, and so on; and it may be a sufficient option when such finer control is required.

DTrace can also be used in a Solaris Zones environment for monitoring multiple zones simultaneously from the global zone, even while the nonglobal zones are rebooted. Administrators and users in the nonglobal zone may not necessarily be aware that global-zone DTrace monitoring is active and, even if they were, cannot do anything to stop it.

2. Provided that it is legal for you to do so; privacy laws differ between countries.

HIDS

DTrace could be used as part of a HIDS to detect and report suspicious activity on the system it is running on. The capabilities with DTrace are as follows.

Detection: Anything that can be traced can be detected. This includes logins, command execution, file system activity, and network I/O.

Reporting: DTrace can output to a log that is post processed by additional reporting software; or, the `system()` action can run shell commands that do the reporting.

Although DTrace can provide a form of HIDS, there are usually compelling reasons to perform intrusion detection using Network Intrusion Detection Systems (NIDS) instead, in particular, monitoring numerous hosts by inspecting network traffic from a single tap port on a switch or router. One possible advantage of a DTrace-based HIDS is that it can be selective with the events it monitors, rather than inspecting every packet (which may become impractical for high-load environments). For example, instead of inspecting every packet to identify both accepted and rejected TCP connections, DTrace can be used to trace only those events from the kernel TCP/IP stack (if the tcp provider is available, this is trivial to do).

Another possible advantage of a DTrace-based HIDS is the inspection of activity that is encrypted over the wire, including SSH and HTTPS. Since these are decrypted on the server where DTrace is running, it can examine both the plain text from the encrypted sessions and the events that they call.

Environments with higher security requirements may find it desirable to run both NIDS for LAN-wide intrusion detection and DTrace-based HIDS where needed.

Similar to auditing, a Solaris zones environment allows a DTrace-based HIDS to be run in the global zone to monitor all nonglobal zones. Even if a nonglobal zone is compromised, the intruder cannot stop the global-zone HIDS.

Policy Enforcement

With the destructive pragma, DTrace can be used ad hoc to help enforce a security policy, such as raising signals to kill processes, execute commands, or even panic the system. The term *ad hoc* is used for the same reasons that DTrace isn't entirely suitable for security audit logs (described earlier): There are scenarios where DTrace could drop events or stop running entirely. Although not ideal, ad hoc enforcement of security may be the best option available if system vulnerabilities are discovered and a security patch is not yet available.

When using DTrace for enforcement, you should take care regarding the timing of the enforcement action. The DTrace `raise()` built-in is immediate, whereas the `system()` built-in is not—and is executed asynchronously sometime after the event (switchrate tunable). The time delta may allow an attacker to reach their goal, such as completing a connection or modifying a file.

Similar to auditing, a Solaris zones environment may allow a DTrace-based security policy to be executed in the global zone, such as preventing nonglobal zones from being able to place network devices in promiscuous mode (see the `nosnoopforyou.d` script).

Privilege Debugging

Apart from the Solaris privileges required to run DTrace, there are numerous other fine-grained privileges that may be used by application software. The `ppriv(1)` command allows privilege usage by software to be debugged for troubleshooting assignment issues and for determining which privileges are required. DTrace can also help, with the sdt provider probes `priv-ok` and `priv-err` for tracing successful and unsuccessful privilege checks. Here they are traced via a one-liner on Oracle Solaris, showing the privilege number (`arg0`) and process name:

```
solaris# dtrace -n 'sdt:::priv-* { printf("for %d by %s\n", arg0, execname); }'
dtrace: description 'sdt:::priv-* ' matched 6 probes
  0  13405          priv_policy_ap:priv-ok for 24 by ssh
  0  13405          priv_policy_ap:priv-ok for 24 by ssh
  0  13405          priv_policy_ap:priv-ok for 24 by ssh
  0  13403        priv_policy_only:priv-ok for 42 by ssh
  0  13405          priv_policy_ap:priv-ok for 42 by ssh
```

This has caught the execution of successful privilege checks by an `ssh` process (which was performing an outbound connection). The privilege codes are in `/usr/include/sys/priv_const.h`:

```
#define PRIV_NET_ACCESS         24
#define PRIV_PROC_SETID         42
```

And are described in the `privileges(5)` man page. To see why `ssh` is accessing these privileges, the user stack trace can be examined by including the `ustack()` action to see the path through the `ssh` source.

A privilege debugging tool has been written that uses these probes, `privdebug.pl` (the source is not included here). It is available to download from the "Privilege Debugging Tool" page[3] from the OpenSolaris security group site, which also links to a white paper[4] by the authors to explain privilege debugging using DTrace. The tool can be used to identify which privileges a body of software accesses, even if that software includes multiple process IDs. Here the privileges accessed during startup of `proftpd` (FTP daemon) are traced:

```
solaris# privdebug.pl -n proftpd
STAT PRIV
USED net_access
USED net_access
USED net_access
USED proc_setid
USED proc_setid
USED proc_setid
USED proc_setid
USED proc_fork
USED net_access
USED net_access
USED net_privaddr
```

This can be useful to determine what privileges software *should* have under normal operation, by running it as root and tracing what actual privileges it used; then, those privileges can be granted as a minimum set (not root).[5] If that software is later compromised (vulnerability), then only the minimum set of privileges have been compromised. Also, the removal of unnecessary privileges may be an effective workaround for existing vulnerabilities that require those privileges to work.

The `-v` option to `privdebug.pl` prints more details; here, the `in.telnetd` daemon was traced as a telnet connection was established:

```
solaris# privdebug.pl -n in.telnetd -f -v
STAT TIMESTAMP        PPID  PID   PRIV            CMD
USED 1231183251124451 238   7115  sys_audit       in.telnetd
USED 1231183251139719 238   7115  sys_audit       in.telnetd
USED 1231183251612259 238   7115  proc_fork       in.telnetd
USED 1231183251974167 7115  7116  proc_exec       in.telnetd
USED 1231183472328575 238   7115  proc_fork       in.telnetd
USED 1231183472556716 7115  7117  proc_exec       in.telnetd
USED 1231183478414533 238   7115  proc_fork       in.telnetd
USED 1231183482742793 7115  7118  file_dac_write  in.telnetd
USED 1231183504062754 7115  7118  proc_exec       in.telnetd
```

3. *http://hub.opensolaris.org/bin/view/Community+Group+security/privdebug*
4. See *www.sun.com/blueprints/0206/819-5507.pdf* by Darren Moffat and Glenn Brunette.
5. This is discussed further in "Limiting Service Privileges in the Solaris 10 Operating System," currently at *www.sun.com/blueprints/0505/819-2680.pdf*.

Similar observability is available on FreeBSD via the priv provider, which has the `priv_ok` and `priv_err` probes. The first argument to these probes (`arg0`) is the privilege number from `/usr/src/sys/sys/priv.h`. Here's an example of using this to trace an `ssh` login, showing the process name, privilege number, and kernel stack trace:

```
freebsd# dtrace -n 'priv:::priv* { printf("%s, priv %d", execname, arg0); stack(); }'
dtrace: description 'priv:::priv* ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  38015               priv_check:priv_ok sshd, priv 160
              kernel`priv_check_cred+0xee
              kernel`fork1+0x59e
              kernel`fork+0x29
              kernel`syscall+0x3e5
              kernel`0xc0bc2030

  0  38015               priv_check:priv_ok sshd, priv 326
              kernel`priv_check_cred+0xee
              kernel`priv_check+0x26
              kernel`vn_stat+0x198
              kernel`vn_statfile+0x15a
              kernel`kern_fstat+0x83
              kernel`fstat+0x27
              kernel`syscall+0x3e5
              kernel`0xc0bc2030
[...]

  0  38014              priv_check:priv_err sshd, priv 50
              kernel`priv_check_cred+0xbf
              kernel`setuid+0xca
              kernel`syscall+0x3e5
              kernel`0xc0bc2030

  0  38014              priv_check:priv_err sshd, priv 51
              kernel`priv_check_cred+0xbf
              kernel`seteuid+0xca
              kernel`syscall+0x3e5
              kernel`0xc0bc2030
```

The output has been truncated because many privileges were checked during the normal operation of `sshd`, firing the `priv_ok` probe. Errors for privileges 50 and 51 were also traced, which are for credential checks for the `setuid()` and `seteuid()` calls.

Reverse Engineering

If you find an unknown program running that you suspect to be malware or spyware, DTrace can be used to examine the operation of the software to confirm. This can include examining which files are being opened, read, and written; what network I/O is being performed to which remote hosts and ports; what data is being sent; and the user code responsible for these events including the user stack back trace. This analysis can begin with the syscall provider, as demonstrated in `networkwho.d` in the "Scripts" section and also shown in *In Phrack Magazine* issue 63, "Analyzing Suspicious Binary Files and Processes" by Boris Loza.[6]

Scripts

Table 11-2 summarizes the scripts that follow and the providers they use.

The fbt provider is considered an "unstable" interface, because it instruments a specific operating system version. The fbt provider-based scripts is this chapter were written for a particular version of Oracle Solaris and will require modifications to execute properly on other kernel versions. See Chapter 12, Kernel, for more discussion about using the fbt provider.

sshkeysnoop.d

As an example of sniffing, the `sshkeysnoop.d` program traces keystrokes from any client `ssh` command running on the system. These are traced as the keystrokes are entered, where they can be examined as plain text before encryption is applied.

Table 11-2 Security Script Summary

Script	Type	Description	Provider
`sshkeysnoop.d`	Sniffer*	Shows `ssh` command keystrokes	syscall
`shellsnoop`	Sniffer*	Watches other shell sessions	syscall
`keylatency.d`	Sniffer*	Measures inter-keystroke latency	syscall
`cuckoo.d`	Sniffer*	Captures serial line sessions	fbt
`watchexec.d`	HIDS	Watches for new command executions and then alerts	syscall
`nosetuid.d`	Enforcement	Only allow specified UID to `setuid()` to root	syscall
`nosnoopforyou.d`	Enforcement	Prevents promiscuous mode on network interfaces	fbt
`networkwho.d`	Reverse engineering	Shows the user stack trace for network I/O	syscall

* These tools are included here for study purposes only. Because they can examine user-land data, including keystrokes and passwords, you should consider privacy concerns (including laws!) before using them.

6. See *www.phrack.com/issues.html?issue=63&id=3#article*, which references Brendan Gregg.

Script

This script is written for the current version of OpenSSH shipped with Solaris, where the `ssh` program reads keystrokes by opening and reading from `/dev/tty`. This is traced by watching `ssh` call `open()` on `/dev/tty` and recording the file descriptor for later checking with the `read()` syscall:

```
#!/usr/sbin/dtrace -s
/*
 * sshkeysnoop.d - A program to print keystroke details from ssh.
 *                 Written in DTrace (Solaris 10 build 63).
 *
 * WARNING: This is a demonstration program, please do not use this for
 * illegal purposes in your country such as breeching privacy.
[...truncated...]
 */

#pragma D option quiet

/*
 * Print header
 */
dtrace:::BEGIN
{
        /* print header */
        printf("%5s %5s %5s %5s  %s\n","UID","PID","PPID","TYPE","TEXT");
}

/*
 * Print ssh execution
 */
syscall::exec*:return
/execname == "ssh"/
{
        /* print output line */
        printf("%5d %5d %5d %5s  %s\n\n", curpsinfo->pr_euid, pid,
            curpsinfo->pr_ppid, "cmd", stringof(curpsinfo->pr_psargs));
}

/*
 * Determine which fd is /dev/tty
 */
syscall::open*:entry
/execname == "ssh"/
{
        self->path = arg0;
}

syscall::open*:return
/self->path && copyinstr(self->path) == "/dev/tty"/
{
        /* track this syscall */
        self->ok = 1;
}

syscall::open*:return { self->path = 0; }

syscall::open*:return
/self->ok/
{
        /* save fd number */
        self->fd = arg0;
```

```
}

/*
 * Print ssh keystrokes
 */
syscall::read*:entry
/execname == "ssh" && arg0 == self->fd/
{
        /* remember buffer address */
        self->buf = arg1;
}

syscall::read*:return
/self->buf != NULL && arg0 < 2/
{
        this->text = (char *)copyin(self->buf, arg0);

        /* print output line */
        printf("%5d %5d %5d %5s  %s\n", curpsinfo->pr_euid, pid,
            curpsinfo->pr_ppid, "key", stringof(this->text));
        self->buf = NULL;
}
```

Script sshkeysnoop.d

Wildcards have been used in probe names for this to work on different operating systems (for example, `open64()` on Solaris, `open_nocancel()` on Mac OS X); however, this may match unwanted syscalls as well. If this becomes a problem (since syscalls are added over time), the script can be fine-tuned to match them explicitly.

Example

To demonstrate this script, it was executed in a lab environment where a test account had been created with the username testuser and the password secret123. (Running this on a production system might be an invasion of user's privacy.) While tracing, an outbound `ssh` session was executed to log in to this test account on a remote host:

```
# sshkeysnoop.d
  UID   PID  PPID  TYPE  TEXT
    0 30040 30032   cmd  ssh testuser@mars

    0 30040 30032   key  s
    0 30040 30032   key  e
    0 30040 30032   key  c
    0 30040 30032   key  r       <--- password
    0 30040 30032   key  e
    0 30040 30032   key  t
    0 30040 30032   key  1
    0 30040 30032   key  2
    0 30040 30032   key  3
    0 30040 30032   key

    0 30040 30032   key  l       <--- command line keystrokes
    0 30040 30032   key  s
```

continues

```
    0 30040 30032   key
    0 30040 30032   key  -
    0 30040 30032   key  l
    0 30040 30032   key
^C
```

The username, host, and password are all visible in the output. After logging in, the testuser executed the `ls -l` command. The output of that command is not visible, since this script is only tracing the input keystrokes (see `shellsnoop`).

shellsnoop

Command-line activity from operating system shells can be examined with `shellsnoop`, which traces keystroke reads and `STDOUT` writes from any of the known running shells (`sh`, `ksh`, `bash`, and so on). This capability is similar to the `ttywatcher` tool (pre-DTrace), which monitored sessions via their terminal interface.

Script

The key parts of the `shellsnoop` script are given next. (The full script is in the DTraceToolkit.)

To support command-line options (such as `-q` for quiet mode, demonstrated later), `shellsnoop` is implemented as a DTrace script encapsulated inside a shell script. The `getopts` function is used in the shell to process options and set variables, which are then passed to DTrace:

```
#!/usr/bin/sh
[...]
while getopts dhp:qsu:v name
do
        case $name in
        d)      opt_debug=1 ;;
        p)      opt_pid=1; pid=$OPTARG ;;
[...]
#################################
# --- Main Program, DTrace ---
#
dtrace -n '
 /*
  * Command line arguments
  */
 inline int OPT_debug   = '$opt_debug';
 inline int OPT_quiet   = '$opt_quiet';
 inline int OPT_pid     = '$opt_pid';
[...]
```

shellsnoop traces text originating from the shell (the output of built-ins, for example), as well as text from any subcommands. For example, if the user runs ls -l, shellsnoop must be tracing the STDOUT writes from the ls subcommand. Shells fork() and then exec() these commands, so shellsnoop watches for any exec() that begins from a shell and tracks this process ID in the child associative array for later identification:

```
/*
 * Remember this PID is a shell child
 */
syscall::exec*:entry
/execname == "sh"   || execname == "ksh"  || execname == "csh"  ||
 execname == "tcsh" || execname == "zsh"  || execname == "bash"/
{
        child[pid] = 1;
[...]
}
```

Shell keystrokes and built-in output are traced via read() and write() syscalls from processes with a shell name to file descriptors between 0 and 2 (covers STDIN, STDOUT, STDERR):

```
/*
 * Print shell keystrokes
 */
syscall::write:entry, syscall::read:entry
/(execname == "sh"   || execname == "ksh"  || execname == "csh"  ||
  execname == "tcsh" || execname == "zsh"  || execname == "bash")
 && (arg0 >= 0 && arg0 <= 2)/
{
        self->buf = arg1;
}
[...]
syscall::write:return, syscall::read:return
/self->buf && child[pid] == 0 && OPT_quiet == 0/
{
        this->text = (char *)copyin(self->buf, arg0);
        this->text[arg0] = '\'\\0\'';

        printf("%5d %5d %8s %3s  %s\n", pid, curpsinfo->pr_ppid, execname,
            probefunc == "read" ? "R" : "W", stringof(this->text));
}
syscall::write:return
/self->buf && child[pid] == 0 && OPT_quiet == 1/
{
        this->text = (char *)copyin(self->buf, arg0);
        this->text[arg0] = '\'\\0\'';
        printf("%s", stringof(this->text));
}
syscall::read:return
/self->buf && execname == "sh" && child[pid] == 0 && OPT_quiet == 1/
{
        this->text = (char *)copyin(self->buf, arg0);
        this->text[arg0] = '\'\\0\'';
        printf("%s", stringof(this->text));
}
```

The first output block (lines 186 to 194) is for the default output of `shellsnoop`, which prints columns of details including the text. Line 189 (and 199 and 206) places a `NULL` character to terminate the string, which should read `this->text[arg0] = '\0';`, however, because this entire DTrace script is encapsulated in quote marks, shell escaping was required.

The second output block (lines 195 to 210) is for quiet mode, which only prints the characters that the shell is printing, allowing the shell session to be mirrored by `shellsnoop` (demonstrated later).

The third block (lines 202 to 208) is a special case for the Bourne shell (`sh`) during quiet mode so that the input commands can be seen (they aren't echoed out using `write()`).

Subcommand output is identified by STDOUT or STDERR writes from processes identified by the `child` array, populated earlier:

```
215   /*
216    * Print command output
217    */
218   syscall::write:entry, syscall::read:entry
219   /child[pid] == 1 && (arg0 == 1 || arg0 == 2)/
220   {
221           self->buf = arg1;
222   }
[...]
233   syscall::write:return, syscall::read:return
234   /self->buf && OPT_quiet == 0/
235   {
236           this->text = (char *)copyin(self->buf, arg0);
237           this->text[arg0] = '\'\\0\'';
238
239           printf("%5d %5d %8s %3s  %s", pid, curpsinfo->pr_ppid, execname,
240               probefunc == "read" ? "R" : "W", stringof(this->text));
241
242           /* here we check if a newline is needed */
243           this->length = strlen(this->text);
244           printf("%s", this->text[this->length - 1] == '\'\\n\'' ? "" : "\n");
245           self->buf = 0;
246   }
247   syscall::write:return, syscall::read:return
248   /self->buf && OPT_quiet == 1/
249   {
250           this->text = (char *)copyin(self->buf, arg0);
251           this->text[arg0] = '\'\\0\'';
252           printf("%s", stringof(this->text));
253           self->buf = 0;
254   }
```

The default output prints columns of details and the text and then checks whether the text was already terminated with a newline, adding one if needed. The quiet mode output simply prints the text as is.

Examples

Examples include systemwide sniffing and process watching.

Systemwide Sniffing

The `shellsnoop` program watches all shell sessions systemwide by default. Here it is executed with the `-s` option to show the system time in microseconds with each shell event traced:

```
# shellsnoop -s
TIME(us)          PID  PPID       CMD DIR  TEXT
272520392705   30032 30029      bash   W  l
272520552661   30032 30029      bash   R  s
272520552787   30032 30029      bash   W  s          <--- keystrokes
272520599003   30032 30029      bash   R
272520599073   30032 30029      bash   W
272520739499   30032 30029      bash   R  -
272520739616   30032 30029      bash   W  -
272520836096   30032 30029      bash   R  1
272520836166   30032 30029      bash   W  1
272520987743   30032 30029      bash   R
272520987811   30032 30029      bash   W

272520994415   30096 30032        ls   W  a
272520994444   30096 30032        ls   W  bin
272520994475   30096 30032        ls   W  boot
272520994509   30096 30032        ls   W  dev
272520994537   30096 30032        ls   W  devices    <--- command output
272520994565   30096 30032        ls   W  etc
272520994593   30096 30032        ls   W  export
272520994623   30096 30032        ls   W  home
272520994650   30096 30032        ls   W  kernel
272520994803   30096 30032        ls   W  lib
272520994893   30096 30032        ls   W  mnt
272520994939   30096 30032        ls   W  net
[...]
```

While tracing, an `ls -1` command was executed to list files. Both the reads and echoed writes for the keystrokes can be seen, with the exception of the first `l` since `shellsnoop` began running after that read syscall was entered.

If you look closely at the first column, the time between shell writes and reads (both measured at completion) is the keystroke latency for that read. The keystroke latency between `l` and `s` is 160 ms (272520552661 us to 272520392705 us).

Process Watching

Here `shellsnoop` is directed to watch the keystrokes of a particular shell, with quiet mode set (`-q`) so that only shell text is printed. Both windows of `shellsnoop` and the examined shell are shown here.

Here's the example for window 1, `shellsnoop`:

```
# ps -fp 130138
     UID    PID   PPID   C    STIME TTY         TIME CMD
 brendan 130138 130135   0 01:10:38 pts/2       0:00 -bash
# shellsnoop -q -p 130138
cd /etc
-bash-3.2$ ls -l passwd
-r--r--r--   1 root     root        1061 Apr  6 23:42 passwd
-bash-3.2$ ls -l shadow
-r--------   1 root     sys          508 Apr  6 23:42 shadow
-bash-3.2$ cat -n shadow
cat: cannot open shadow: Permission denied
-bash-3.2$ ^C
```

And here's the text from window 2, showing the target shell:

```
-bash-3.2$ cd /etc
-bash-3.2$ ls -l passwd
-r--r--r--   1 root     root        1061 Apr  6 23:42 passwd
-bash-3.2$ ls -l shadow
-r--------   1 root     sys          508 Apr  6 23:42 shadow
-bash-3.2$ cat -n shadow
cat: cannot open shadow: Permission denied
-bash-3.2$
```

The initial shell prompt wasn't seen by `shellsnoop` because it was printed before tracing began, but after that, `shellsnoop` has mirrored the target shell perfectly. (It helps that this is a single CPU system, and the output doesn't get scrambled slightly because of DTrace reading through the different CPU switch buffers.)

It can be a little spooky to be watching another user's session on your own screen, including editor sessions (`vi(1)` or `vim(1)`) where the cursor dances around the screen modifying text.

keylatency.d

Inter-keystroke latency is the time between keystrokes, which differs based on which keys are being pressed, the distance between those keys (finger travel), the user's keyboard skill, and other typing characteristics. This is sometimes studied in security for improving brute-force password attacks. If keystroke latency can be measured while a user types a secret password, then this information may be used to infer which are less likely keystroke transitions and remove them from a brute-force search. The `keylatency.d` script measures the keystroke latency time.

Script

The `keylatency.d` script watches keystrokes by tracing `read()` syscalls from STDIN. It contains a string `TARGET`, which can be adjusted so that other processes are traced (for example, `vi`).

```
 1  #!/usr/sbin/dtrace -s
 2
 3  #pragma D option quiet
 4
 5  /* process name to monitor */
 6  inline string TARGET = "bash";
 7
 8  self string lastkey;
 9
10  dtrace:::BEGIN
11  {
12          printf("Tracing %s keystrokes...  Hit Ctrl-C to end.\n", TARGET);
13  }
14
15  syscall::read:entry
16  /execname == TARGET && arg0 == 0/
17  {
18          self->buf = arg1;
19          self->start = timestamp;
20  }
21
22  syscall::read:return
23  /self->buf && arg0 == 1/
24  {
25          this->latency = timestamp - self->start;
26          this->key = stringof((char *)copyin(self->buf, arg0));
27          this->key = this->key == "\r" ? "NL" : this->key;       /* return */
28          this->key = this->key == "\t" ? "TAB" : this->key;      /* tab */
29          this->key = this->key == "\177" ? "BS" : this->key;     /* backspace */
30          @a[self->lastkey != NULL ? self->lastkey : " ", this->key] =
31              avg(this->latency);
32          @c[self->lastkey != NULL ? self->lastkey : " ", this->key] = count();
33          self->lastkey = this->key;
34          self->start = 0;
35  }
36
37  syscall::read:return /self->buf/ { self->buf = 0; self->start = 0; }
38
39  dtrace:::END
40  {
41          normalize(@a, 1000000);
42          printf("Average Keystroke Latency for %s processes (ms):\n\n", TARGET);
43          printf("%34s %8s\n", "LATENCY", "COUNT");
44          printa("%16s -> %3s %10@d %@8d\n", @a, @c);
45  }
```

Script keylatency.d

Lines 27 to 29 convert some of the special characters into strings so that when printed they don't mess up the output.

Example

The script was executed while regular commands were typed in another bash shell:

```
# keylatency.d
Tracing bash keystrokes...  Hit Ctrl-C to end.
^C
Average Keystroke Latency for bash processes (ms):

                           LATENCY     COUNT
               h ->   t         13         1
               b ->   s         17         2
               i ->   o         29         1
               c ->   d         33         8
                 ->   s         41         3
               s ->   t         42         3
               h ->   e         43         2
                 ->   /         45         7
               - ->   l         48         9
               t ->   h         54         2
[...output truncated...]
               - ->   x        489         1
              NL ->   c        503         9
               t ->  BS        599         2
              NL ->   l        651         9
              NL ->   j        696         3
              BS ->   l        949         2
              NL ->   t        978         2
              NL ->   d       1279         2
```

One of the fastest keystroke latencies is the transition from `-` to `l`, typed frequently as `ls -l`. The slowest transitions are those after new lines, because this includes think time.

The fastest transition caught was from `h` to `t` at only 13 milliseconds. (This may be a giveaway that the text was typed on a dvorak-layout keyboard.)

cuckoo.d

In the book *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*, Clifford Stoll attached an array of TeleType printers to modem-attached serial lines to capture activity from a remote cracker (black hat hacker). If DTrace had been available to Clifford, capturing session data across the system would have been much easier (and would not have required borrowing so many TeleType printers and terminals). This script does this for Solaris, capturing serial output and displaying it with user and process ID details.

This script is based on the unstable fbt provider and may work only on a particular version of Oracle Solaris. It's included as an example of kernel sniffing capabilities, whether it executes or not; for it to keep working, it will need to be updated to match changes in the kernel code.

Script

This script traces the `cnwrite()` function and pulls in the character data from user-land (assuming it is a UIO_USERSPACE `uio`, which could be tested in a predicate if desired). There may be other ways to do this as well, such as tracing `async_txint()` or `cdev_write()`.

```
 1  #!/usr/sbin/dtrace -s
 2
 3  #pragma D option quiet
 4  #pragma D option switchrate=10hz
 5
 6  dtrace:::BEGIN
 7  {
 8          printf("%-20s %6s %6s %6s %s\n", "TIME", "PID", "PPID", "UID", "TEXT");
 9  }
10
11  fbt::cnwrite:entry
12  {
13          this->iov = args[1]->uio_iov;
14          this->len = this->iov->iov_len;
15          this->text = stringof((char *)copyin((uintptr_t)this->iov->iov_base,
16              this->len));
17          this->text[this->len] = '\0';
18
19          printf("%-20Y %6d %6d %6d %s\n", walltimestamp, pid, ppid, uid,
20              this->text);
21  }
```
Script cuckoo.d

As with `shellsnoop`, strings are manually `NULL` terminated to the known length (line 17).

Example

The echoed keystrokes and command outputs in the following were traced from a serial session (connected via the service processor). This script could be enhanced: `curpsinfo->pr_ttydev` could be printed so that different serial sessions can be differentiated; and as with `shellsnoop`, a quiet mode could be implemented to only print the seen text, mirroring the serial display.

```
# cuckoo.d
TIME                    PID   PPID    UID TEXT
2010 Jun  7 12:49:30  30557  30554      0 d
2010 Jun  7 12:49:30  30557  30554      0 a
2010 Jun  7 12:49:30  30557  30554      0 t
2010 Jun  7 12:49:30  30557  30554      0 e
2010 Jun  7 12:49:30  30557  30554      0

2010 Jun  7 12:49:30  30629  30557      0 Mon Jun  7 12:49:30 UTC 2010

2010 Jun  7 12:49:30  30557  30554      0 lox#
2010 Jun  7 12:49:31  30557  30554      0 l
```
continues

```
2010 Jun  7 12:49:31  30557  30554      0 s
2010 Jun  7 12:49:31  30557  30554      0
2010 Jun  7 12:49:31  30557  30554      0 -
2010 Jun  7 12:49:32  30557  30554      0 l
2010 Jun  7 12:49:32  30557  30554      0

2010 Jun  7 12:49:32  30630  30557      0 total 696

2010 Jun  7 12:49:33  30630  30557      0 -rw-------   1 root     root       32 31 Ap
r  6 23:54 akworAAAEhadig

2010 Jun  7 12:49:33  30630  30557      0 -rw-------   1 root     root       26 02 Ap
r  6 23:55 akworBAAFhadig
[...]
```

watchexec.d

This is an example of an intrusion detection script. The execution of binaries are traced, and any that are not recognized based on a hard-coded "allow" list will generate alerts. The alerts are reported by a custom shell wrapper, which takes the action desired: populate a log, send e-mail, send an SNMP trap, and so on.

Script

The script watches the `exec()` syscall variants and checks whether the executable is in a hard-coded allow list in the script. If not, a shell command is executed to perform the report, which is handed the information including the executable path as shell arguments:

```
#!/usr/sbin/dtrace -s

#pragma D option destructive
#pragma D option quiet

inline string REPORT_CMD = "/usr/local/bin/reporter.sh";

dtrace:::BEGIN
{
      /*
       * Ensure this contains all the reporting commands,
       * otherwise this script will be a feedback loop:
       */
      ALLOWED[REPORT_CMD] = 1;
      ALLOWED["/bin/sh"] = 1;

      /*
       * Commands to allow.
       * Example list (from Solaris) in alphabetical order:
       */
      ALLOWED["/bin/bash"] = 1;
      ALLOWED["/lib/svc/bin/svcio"] = 1;
      ALLOWED["/sbin/sh"] = 1;
      ALLOWED["/usr/apache2/current/bin/httpd"] = 1;
      ALLOWED["/usr/bin/basename"] = 1;
      ALLOWED["/usr/bin/cat"] = 1;
```

```
        ALLOWED["/usr/bin/chmod"] = 1;
        ALLOWED["/usr/bin/chown"] = 1;
        ALLOWED["/usr/bin/grep"] = 1;
        ALLOWED["/usr/bin/head"] = 1;
        ALLOWED["/usr/bin/ls"] = 1;
        ALLOWED["/usr/bin/pgrep"] = 1;
        ALLOWED["/usr/bin/pkill"] = 1;
        ALLOWED["/usr/bin/ssh"] = 1;
        ALLOWED["/usr/bin/svcprop"] = 1;
        ALLOWED["/usr/bin/tput"] = 1;
        ALLOWED["/usr/bin/tr"] = 1;
        ALLOWED["/usr/bin/uname"] = 1;
        ALLOWED["/usr/lib/nfs/mountd"] = 1;
        ALLOWED["/usr/lib/nfs/nfsd"] = 1;
        ALLOWED["/usr/sfw/bin/openssl"] = 1;
        ALLOWED["/usr/xpg4/bin/sh"] = 1;

        printf("Reporting unknown exec()s to %s...\n", REPORT_CMD);
}

syscall::exec*:entry
/ALLOWED[copyinstr(arg0)] != 1/
{
        /*
         * Customize arguments for reporting command:
         */
        system("%s %s %d %d %d %Y\n", REPORT_CMD, copyinstr(arg0),
            uid, pid, ppid, walltimestamp);
}
```

Script watchexec.d

The arguments printed are a starting point, which can be enhanced. The current working directory could be added (cwd), although directories (and executables) may contain whitespace, which will need to be considered if treating the arguments to the reporter as whitespace delimited. Another useful addition would be the return value of `exec()` by tracing `exec:return` to see whether it was successful.

The `/usr/local/bin/reporter.sh` script performs the reporting. It may log to a file, write to syslog, send an e-mail, send an SNMP trap, write to a database, or some combination of these. Examples are beyond the scope of this book; as a starting point, the following `reporter.sh` takes the arguments and simply appends them to a log file:

```
reporter.sh:
        #!/bin/sh

        echo "$*" >> /var/log/execlog.txt
```

This could be executed as a trial run to check what is identified and to improve the `ALLOWED` list, before using a `reporter.sh` that sends an e-mail or an SNMP trap.

Either `watchexec.d` or `reporter.sh` could be easily modified to support other policies as well, such as monitoring for activity outside of work hours, from certain IP addresses, and so on. You can be as creative as you like because of the flexibility of DTrace and its ability to collect virtually any kind of information on the system.

Example

As a test, `watchexec.d` was run with the simple logging `reporter.sh`, while a user logged in and ran a couple of binary executables from their home directory:

```
# watchexec.d
Reporting unknown exec()s to /usr/local/bin/reporter.sh...

# cat /var/log/execlog.txt
./a.out 1001 8919 8911 2009 Sep 19 19:32:55
./ls 1001 8919 8911 2009 Sep 19 19:33:07
/etc/dhcp/eventhook 0 8852 8851 2009 Sep 19 19:35:27
```

The log shows that the user with UID 1001 ran an unknown executable `./a.out`, and another called `./ls`. The invocation of `ls` is suspicious since it is usually called via the shell PATH as `/usr/bin/ls`; so although this may be innocent (`cd /usr/bin; ./ls`), it may also be a malicious binary that has been renamed `ls`.

`watchexec.d` is an example DTrace component of a simple intrusion detection system. Additional components needed include the `reporter.sh` script and a means to start and restart `watchexec.d`, considering that DTrace can abort tracing and this needs to keep running. It may also make sense for `watchexec.d` to call `reporter.sh` on startup from `dtrace:::BEGIN` so that if the script has began restarting for some reason (systemic unresponsiveness), that would also be reported.

nosetuid.d

This is an example of ad hoc security enforcement. Here the `setuid()` syscall is traced and blocked based on a simple security policy: Only the allowed UID can become UID 0, "root." `setuid()` is used by software such as `su(1M)` (set user) and `sudo(8)` to become a different user, usually root (UID 0), after authenticating.

If a security vulnerability was found that allows nonroot users to `setuid()` to root without the correct password, a script such as this could provide a form of defense while waiting for the operating system vendor to provide a patch.

Script

This script raises the `KILL` signal to processes using the `raise()` action and because of this requires the destructive pragma. Be sure that you understand the implications before running this script; as a trial, line 22 could be deleted and the script executed to test whether it would have killed any normal application activity by mistake.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option destructive

inline int ALLOWED_UID = 517;

dtrace:::BEGIN
{
      printf("Watching setuid(), allowing only uid %d...\n", ALLOWED_UID);
}

/*
 * Kill setuid() processes who are becomming root, from non-root, and who
 * are not the allowed UID.
 */
syscall::setuid:entry
/arg0 == 0 && curpsinfo->pr_uid != 0 && curpsinfo->pr_uid != ALLOWED_UID/
{
      printf("%Y KILLED %s %d -> %d\n", walltimestamp, execname,
          curpsinfo->pr_uid, arg0);
      raise(9);
}
```

Script nosetuid.d

Example

While running the script, the user brendan (UID 1001) attempts to `su` to root (UID 0):

```
$ id
uid=1001(brendan) gid=1(other)
$ su -
Password:
$ id
uid=1001(brendan) gid=1(other)
```

The `su` command has been killed, and the uid was not changed to root. The root user running the `nosetuid.d` script saw the following:

```
# nosetuid.d
Watching setuid(), allowing only uid 517...
2009 Sep 19 06:51:57 KILLED su 1001 -> 0
2009 Sep 19 06:57:57 KILLED sendmail 25 -> 0
```

Not only has it killed the `su` command, but it also saw and killed a `sendmail` command by mistake! This may be a good example of why actions such as `raise()` require the destructive pragma—to indicate that the script has the capability to cause harm.

nosnoopforyou.d

The network packet sniffer utility on Solaris is `snoop(1M)`. As a more complex example of ad hoc enforcement, the `nosnoopforyou.d` script prevents users from performing network sniffing on Solaris, such as by using `snoop(1M)`. This example is more complex, because it identifies network sniffing by watching for interfaces being placed in promiscuous mode inside the kernel, rather than matching on the `execname` "snoop" (since users could then just copy and rename `snoop(1M)`). By matching inside the kernel, even if users wrote and compiled their own userland software to perform network sniffing, it would still be identified and killed.

Warning

Not only is this an fbt provider–based script and therefore prone to misexecute on different kernel versions, it also uses the destructive pragma so that it can raise the `KILL` signal to processes. So, if it stops working properly and misidentifies processes, it could kill them and cause harm to the system.

Script

This script would have been much easier had the kernel promiscuous functions executed in the same thread as the user-land code. It would then be a matter of just tracing the right function and raising a signal. Instead, those functions are executed by a kernel task queue thread, after the user thread has stepped off-CPU. This script traces the event before enqueuing the task, while the user thread is still on-CPU and can be killed:

```
#!/usr/sbin/dtrace -Cs

#pragma D option quiet
#pragma D option destructive

/* /usr/include/sys/dlpi.h: */
#define     DL_PROMISCON_REQ 0x1f

dtrace:::BEGIN
{
      trace("Preventing promiscuity...\n");
}

fbt::dld_wput_nondata:entry
{
      this->mp = args[1];
      this->prim = ((union DL_primitives *)this->mp->b_rptr)->dl_primitive;
}

fbt::dld_wput_nondata:entry
/this->prim == DL_PROMISCON_REQ/
{
```

```
23          printf("%Y KILLED %s PID:%d PPID:%d\n", walltimestamp, execname,
24              pid, ppid);
25          /* raise(9); */
26      }
```
Script nosnoopforyou.d

Line 25 has been commented out in this script in case anyone copies and pastes without reading the previous warning. If you understand the warning and want this script to work, remove the comment characters from that line.

Example

In the window1 session, the script was run. In the window2 session, the `snoop(1M)` command was executed, which was immediately killed. Details can be seen in the output of the script:

```
window1# nosnoopforyou.d
Preventing promiscuous mode...
2009 Sep 19 22:04:20 KILLED snoop PID:9273 PPID:9193

window2# snoop
Using device e1000g0 (promiscuous mode)
Killed
window2#
```

networkwho.d

The `networkwho.d` script shows the user-land stack trace when a process performs writes and socket connections so that the code-path to network I/O can be identified. This may be the first of many ways that DTrace can examine the behavior of an unknown binary (malware or spyware) as it executes.

Script

The script takes a PID argument and traces the user-land stack trace whenever it calls `connect()`, `listen()`, `write()`, or `send()`. Since the script may be executed after the `connect()`, it can't rely on tracing it to provide file descriptor details for later filtering with `write()` and `send()`; instead, it traces all `write()` and `send()` calls and prints the file descriptor type:

```
#!/usr/sbin/dtrace -s

#pragma D option defaultargs
#pragma D option switchrate=10hz

dtrace:::BEGIN
/$1 == 0/
```
continues

```
{
        printf("USAGE: networkwho.d PID\n");
        exit(1);
}

syscall::connect:entry,
syscall::listen:entry
/pid == $1/
{
        ustack();
}

syscall::write*:entry,
syscall::send*:entry
/pid == $1/
{
        trace(fds[arg0].fi_fs);
        ustack();
}
```

Script networkwho.d

Example

To demonstrate `networkwho.d`, it was pointed at an active `ssh` process:

```
# networkwho.d 9136
[...]
  0  89023                    write:entry   sockfs
              libc.so.1`__write+0x15
              ssh`packet_write_poll+0x37
              ssh`client_loop+0x47a
              ssh`ssh_session2+0x5c
              ssh`main+0xd9f
              ssh`_start+0x7d
```

The stack trace within `ssh` that is performing the writes has been shown, identifying the code path taken.

Summary

DTrace has some interesting uses for security, including the ability to sniff data at any layer of the software stack, debug the use of system privileges, and examine the operation of suspicious software. DTrace was designed as a debugger that can drop events under load, which makes some uses such as auditing and policy enforcement unreliable using DTrace. It may, however, be better than nothing at all. This chapter demonstrated these uses with several D scripts.

12

Kernel

The operating system kernel is the software at the heart of a system, managing system resources and user processes. It has historically been difficult to observe as it executes in a protected context, beyond the reach of process debuggers. DTrace provides custom visibility into kernel operations, allowing you to answer questions such as the following.

Where is the kernel spending time consuming CPU cycles?

What kernel memory allocations are occurring, and for which segments?

When are functions executing? And with what arguments?

Why are functions being executed? What is their stack backtrace?

How long does it take to execute kernel functions? On-cpu/off-cpu?

As an example, the following one-liner traces all kernel function calls beginning with `vmem` (kernel virtual memory subsystem), printing a time stamp in nanoseconds for when the function began and finished executing:

```
solaris# dtrace -n 'fbt::vmem_*: { trace(timestamp); }'
dtrace: description 'fbt::vmem_*: ' matched 66 probes
CPU     ID                    FUNCTION:NAME
  0  37099                 vmem_alloc:entry   126065068702911
  0  37100                vmem_alloc:return   126065068705647
  0  36499           vmem_is_populator:entry   126065068779334
  0  36500          vmem_is_populator:return   126065068780982
```

continues

```
  0  42127                   vmem_size:entry   126066151505074
  0  42128                  vmem_size:return   126066151526361
[...]
```

This shows when `vmem` calls are executed, as they happen in the kernel. Before DTrace, this type of visibility was impossible; in order to provide real-time debugging, you needed a custom build of the kernel that included extra instrumentation for these function calls.

This chapter introduces kernel analysis using DTrace, providing a suggested strategy for analysis, checklist of common issues, and example DTrace one-liners and scripts. There is also additional discussion for certain topics, including kernel tracing and memory allocation.

The kernel is an advanced topic that cannot be fully explained within a single chapter of this book, so some familiarity with kernel internals is assumed. For reference, see the following:

Solaris Internals (McDougall and Mauro, 2006)

Mac OS X Internals (Singh, 2006)

The Design and Implementation of the FreeBSD Operating System (Neville-Neil and McKusick, 2004)

These are also listed in the bibliography.

Capabilities

Major components of the kernel are pictured in Figure 12-1, all of which may be visible via DTrace depending on the kernel build and the inclusion of symbol information (Solaris and OpenSolaris are usually built to include all function symbols, making everything visible).

Use DTrace to answer questions about an operating system kernel such as the following.

Which system calls are occurring, and by which processes? Where in the user code are they originating (user stack trace)? What arguments are being passed? What is the system call latency?

What calls into the Virtual File System (VFS) layer are occurring? Which files are being read and written?

What virtual memory functions are being called? Why (kernel stack trace)?

What is the kernel load in terms of process and thread creation?

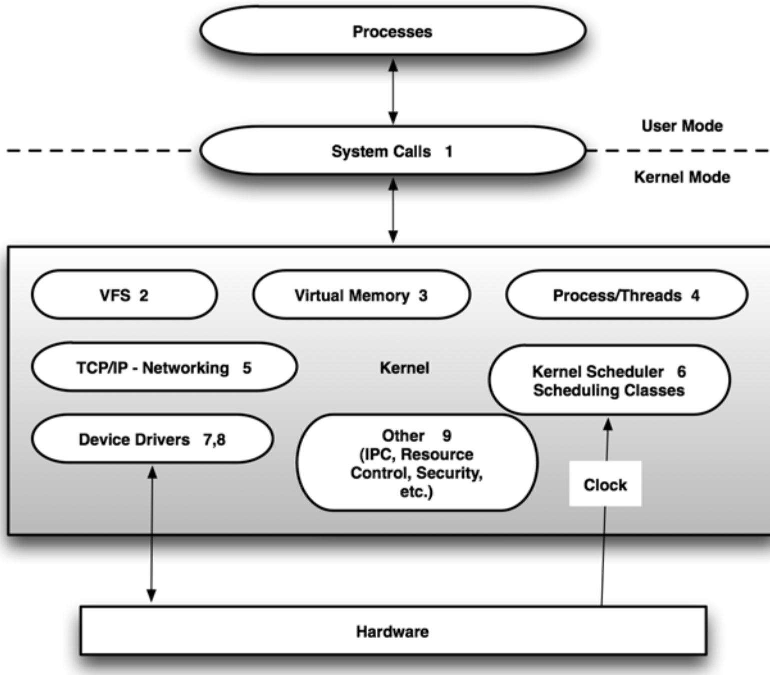


Figure 12-1 Operating system kernel functional diagram

How much of the kernel execution time is in networking code? Which processes are generating the network load?

Which threads are being taken off-CPU, and why? How long are runnable threads waiting for CPU?

What is the interrupt load on CPUs, and from which devices?

Do device drivers encounter errors during boot (anonymous tracing)?

What other areas of the kernel are consuming CPU cycles and memory?

In addition, the following chapters cover topics that are provided by or related to the kernel:

Chapter 4, Disk I/O (includes kernel disk I/O interface and drivers)

Chapter 5, File Systems (these are typically implemented in the kernel)

Chapter 6, Network Lower-Level Protocols (includes the kernel TCP/IP stack)

Chapter 7, Application-Level Protocols (some application protocols are implemented as kernel drivers)

Strategy

To get started using DTrace to examine the operating system kernel, follow these steps:

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow.
2. Instrument and observe the **system calls** (syscall provider). The system call layer is where applications meet the kernel. Observing system calls provides good insight into which underlying kernel subsystems are being most heavily utilized.
3. Check which **stable providers** exist that may help, such as profile, sched, vminfo, and lockstat, and the documentation for these providers. The profile provider is especially effective at identifying why the kernel is on-CPU, as shown in the one-liners. Also check for the existence of sdt provider probes.
4. Before examining kernel function calls with the fbt provider, see what **documentation** exists for the kernel topic of interest (see earlier references).
5. Trace kernel function execution using the **fbt provider**. Finding the best probes to use out of the thousands available can be the real challenge; see Chapter 14, Tips and Tricks, for examples of using `grep(1)`, known workloads and frequency counting. Additional techniques are as follows:
   - 5.1 Examine the **kernel source code**, if available (requires a basic understanding of the kernel programming language, such as C for the kernel). Reading the kernel source not only can find function probes of interest but is the best way to determine the arguments and return value of functions.
   - 5.2 You can navigate the kernel by following program flow, tracing function entry and return probes with the **flowindent** pragma. Another way is to pick deep and logical points in the kernel (such as a device driver performing I/O) and to examine **stack backtraces**, which illustrate the path to that point.
   - 5.3. Familiarize yourself with any **existing kernel statistics**. For example, Solaris has `kstat` (kernel statistics), which can be listed using the `kstat -p` command. Where they are incremented in the kernel source code can be used for navigation through unfamiliar code.
6. If you are not in kernel engineering or don't work for the operating system vendor, consider asking the vendor to provide DTrace scripts for you, proving the business need for the observability.

Table 12-1 Kernel Checklist

Issue	Description
On-CPU	When CPUs are busy in what some tools report as system (%sys) time (the kernel), DTrace can determine which kernel modules and code paths are responsible. Reasons for busy CPUs include the following: • Lock contention (spin) • CPU cross calls • Hot code paths • Memory bus I/O
Off-CPU/ latency	Different types of off-cpu latency can be encountered in the kernel: • Device I/O time (disks, network) • CPU scheduler dispatcher queue latency • Lock wait • Conditional variable wait
Errors	Check whether errors are being encountered and communicated via the appropriate interface.
Configuration	If kernel tuning parameters are set, it can be worthwhile to use DTrace to check that they are taking effect.

Checklist

Consider the checklist of kernel events shown in Table 12-1 that can be examined using DTrace.

Providers

Table 12-2 shows providers of interest when tracing the kernel:

Table 12-2 Providers for Kernel Observability

Provider	Description
syscall	Traces entry and return of operating system calls, arguments, and return values.
profile	Sample kernel activity at a custom rate.
sched	Traces kernel thread scheduler events.

continues

Table 12-2 Providers for Kernel Observability (*Continued*)

Provider	Description
vminfo	Virtual memory statistic probes, based on `vmstat(1M)` statistics.
sysinfo	Kernel statistics probes, based on `mpstat(1M)` statistics.
lockstat	Traces kernel lock events.
sdt	Kernel modules sometimes have interesting sdt probes implemented by the kernel engineer for debugging purposes.
fbt	Traces kernel function execution, arguments, and return values (an unstable interface).

The full reference for provider probes and arguments is in the DTrace Guide[1] and summarized in Appendix C. Additional fbt provider discussion follows.

fbt Provider

The Function Boundary Tracing (fbt) provider instruments kernel function execution, providing probes for kernel function entry and return points. It also provides access to function arguments, return codes, and return instruction offsets. By tracing function entry and return, the elapsed time and on-CPU time during function execution can also be measured.

Listing fbt provider probes on Mac OS X 10.6:

```
macosx# dtrace -ln fbt:::
   ID   PROVIDER            MODULE                          FUNCTION NAME
   41        fbt       mach_kernel                      AllocateNode entry
   42        fbt       mach_kernel                      AllocateNode return
   43        fbt       mach_kernel                            Assert entry
   44        fbt       mach_kernel                            Assert return
   45        fbt       mach_kernel                        BF_decrypt entry
   46        fbt       mach_kernel                        BF_decrypt return
   47        fbt       mach_kernel                        BF_encrypt entry
   48        fbt       mach_kernel                        BF_encrypt return
   49        fbt       mach_kernel                        BF_set_key entry
[...18346 lines truncated...]
```

Stability

The fbt provider is considered an *unstable* interface, meaning that the provider interface (which consists of the probe names and arguments) may be subject to

1. *http://wikis.sun.com/display/DTrace/Documentation*

change between kernel versions. This is because the interface is dynamically constructed based on the thousands of functions that make up the current implementation of the kernel. These kernel functions are subject to change, and when they do, so does the fbt provider.

This means that any DTrace scripts or one-liners based on the fbt provider may be dependent on the kernel version for which they were written. At the very least, fbt-based scripts are unlikely to be portable between Solaris, Mac OS X, and FreeBSD, since the kernels are significantly different. The kernel also changes from version to version for the same operating system, so an fbt-based script written for Solaris 10 update 1 may not work on Solaris 10 update 2 and may not even work after a minor kernel patch on Solaris 10 update 1.

If an fbt-based script has stopped working because of minor kernel changes, it may be that the script can be repaired with equivalent minor changes to match the newer kernel. If the kernel has changed significantly, then the fbt-based script may need to be rewritten entirely. Because of this instability, you should use fbt only when needed. If there are stable providers available that can serve the same role, use those instead. The scripts that use them will not need to be rewritten as the kernel changes.

Because fbt is an unstable interface, these scripts are not guaranteed to work or to be supported by the operating system vendors. Despite the instability, it is still of enormous value that fbt tracing is possible at all, and using it can and has solved countless issues.

The scripts in this book serve as examples of using fbt—not just for how the fbt provider is used in D programs but also for example data that DTrace can make available and showing why that can be useful. If these scripts stop working, you can try fixing them yourself or check for updated versions on the Web (try this book's Web site).

See Chapter 6, Network Lower-Level Protocols, and the discussion around `tcpsnoop.d` as a case study for fbt instability.

Probe Count

The number of probes differs depending on the kernel that is being dynamically instrumented. The following examples compare the available probe count by listing probes and counting lines.

Here's the example for Oracle Solaris Nevada:

```
solaris# dtrace -ln fbt::: | wc -l
   69113
```

Here's the example for Mac OS X Snow Leopard:

```
macos_x# dtrace -ln fbt::: | wc -l
   18356
```

Here's the example for FreeBSD 8.0:

```
freebsd# dtrace -ln fbt::: | wc -l
   37133
```

Regardless of the kernel, there should be at least 10,000 probes available.

Module Name

The kernel module name is the second field of the probe name. Listing these on Solaris yields the following:

```
solaris# dtrace -ln 'fbt:::' | awk '{ print $2":"$3 }' | sort -u
PROVIDER:MODULE
fbt:FX
fbt:FX_DPTBL
fbt:RT
fbt:RT_DPTBL
fbt:SDC
fbt:TS
fbt:TS_DPTBL
fbt:acpi_drv
fbt:acpica
fbt:acpidev
fbt:acpinex
[...165 lines truncated...]
```

This allows all probes from a particular module to be matched—for example, `fbt:zfs::entry` for all the function entry probes from ZFS.

Mac OS X currently doesn't make use of the module field, which only ever contains `mach_kernel`. This doesn't turn out to be much of a problem because of the naming conventions of many kernel modules, which prefix the function name with the module name. For example, finding the HFS (file system) functions on Mac OS X is possible using wildcards, as shown by the following:

```
macos_x# dtrace -ln 'fbt::hfs_*:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 9396        fbt       mach_kernel                  hfs_addconverter entry
 9398        fbt       mach_kernel                          hfs_bmap entry
 9400        fbt       mach_kernel                         hfs_chkdq entry
 9402        fbt       mach_kernel                      hfs_chkdqchg entry
```

```
 9404        fbt       mach_kernel                             hfs_chkiq entry
[...]
```

Arguments and Return Value

The arguments and return value for kernel functions can be inspected on the fbt entry and return probes.

> `fbt:::entry`: The typed arguments are available as `args[0]` ... `args[n]`.
>
> `fbt:::return`: The program counter is `args[0]`; the return value is `args[1]`.

The `arg0` ... `argn` variables are the same but cast as uint64_t (64-bit unsigned integers).

Symbol information is built in to the kernel to allow navigation of typed C structs. This allows any information passed as arguments or return values to be inspected using D language statements that match C. For example, consider the following C code from Oracle Solaris ZFS:

```
uts/common/fs/zfs/arc.c:
[...]
 2247  static void
 2248  arc_get_data_buf(arc_buf_t *buf)
 2249  {
 2250          arc_state_t             *state = buf->b_hdr->b_state;
 2251          uint64_t                size = buf->b_hdr->b_size;
 2252          arc_buf_contents_t      type = buf->b_hdr->b_type;
 2253
 2254          arc_adapt(size, state);
```

The argument to `arc_get_data_buf()` is an `arc_buf_t` pointer, presented in DTrace as `args[0]`. The definition for `arc_buf_t` can be examined to search for members of interest. Another way to find members is to inspect their usage in the function code. In this case, lines 2250 to 2252 show how state, size, and type can be retrieved. The following fetches the size using DTrace:

```
# dtrace -n 'fbt::arc_get_data_buf:entry { trace(args[0]->b_hdr->b_size); }'
dtrace: description 'fbt::arc_get_data_buf:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
 12  48494           arc_get_data_buf:entry              9728
 12  48494           arc_get_data_buf:entry               512
 12  48494           arc_get_data_buf:entry               512
 12  48494           arc_get_data_buf:entry              4608
 12  48494           arc_get_data_buf:entry               512
[...]
```

Data can be fetched in this way from deep within kernel structures. Other examples of retrieving useful data from kernel structures can be found in `/usr/lib/dtrace` translators.

On Solaris systems, you can also determine the arguments passed to a kernel function of interest using the integrated modular debugger, `mdb(1)`, and its built-in commands:

```
solaris# mdb -k
Loading modules: [ unix genunix dtrace specfs ufs sd mpt px ldc ip hook neti
sctp arp usba nca fcp fctl emlxs ssd md lofs zfs random cpc crypto ptm sppp nfs ipc ]
> zfs_read::nm -f ctype
C Type
int (*)(vnode_t *, uio_t *, int, cred_t *, caller_context_t *)
> ::print -t vnode_t
{
    kmutex_t v_lock {
        void *[1] _opaque
    }
    uint_t v_flag
    uint_t v_count
[...]
    char *v_path
[...]
}
```

The `::nm -f ctype` command was executed on `zfs_read` to print its function prototype, which shows arguments and return value as C language data types. This shows that `zfs_read()` returns a pointer to an integer (`int (*)`) and takes four pointers as arguments, each pointing to a kernel data structure (`vnode_t`, and so on).

The `::print -t` command was executed with `vnode_t` to list the structure members along with their data types. This shows that among the many variables stored in a vnode is a character pointer called `v_path`, which is a `NULL`-terminated string containing the cached vnode path name. We can use this information for our DTrace invocation to look at `zfs_read()`:

```
solaris# dtrace -n 'fbt:zfs:zfs_read:entry { @[stringof(args[0]->v_path)] =
count(); }'
dtrace: description 'fbt:zfs:zfs_read:entry ' matched 1 probe
^C

  /scratch/aime/nchand/aime_armix_main/opmn/conf/.formfactor.dnagad01
1
  /scratch/aime/nchand/aime_armix_main/has_work/listener.ora              2
  /scratch/aime/nchand/aime_armix_main/opmn/conf/ons.config.dnagad01              4
  /scratch/aime/nchand/aime_armix_main/rdbms/bin/oracle              4
  /scratch/aime/nchand/aime_armix_main/work/sqlnet.ora              4
```

In the previous example, we enabled the entry point to the kernel `zfs_read()` function, and using the information derived from the `mdb(1)` session, we aggregate on the file path name embedded in the vnode referenced as the first argument. Note we used the DTrace `stringof()` function to treat the character pointer (`v_path`) as a string so that it can be printed.

Kernel Tracing

The fbt provider along with the DTrace `flowindent` option enables a powerful way of tracing kernel function flow. It may at times be interesting to trace the code flow through the kernel for a specific event or system call, for the purpose of timing and profiling, for investigating a problem and needing to know which kernel functions are being called, or perhaps as an exercise in studying kernel internals. Using the syscall provider as the entry point into tracing the kernel is particularly interesting because system calls are the entry point into the kernel from user processes.

The `ktrace.d` script enables tracing the function call flow through the kernel from the entry point of a system call, provided as a command-line argument:

```
#!/usr/sbin/dtrace -s
#pragma D option flowindent

syscall::$1:entry
{
        self->flag = 1;
}
fbt:::
/self->flag/
{
}
syscall::$1:return
/self->flag/
{
        self->flag = 0;
        exit(0);
}
```

Script ktrace.d

Note the use of the macro variable `$1` in the probe function field of the syscall provider entry and return probes, allowing us to specify which system call we want to trace on the command line. Executing this for the `write(2)` system call yields the following:

```
solaris# ./ktrace.d write
dtrace: script './ktrace.d' matched 68135 probes
CPU FUNCTION
  3  -> write
```

continues

```
    3    -> getf
    3      -> set_active_fd
    3      <- set_active_fd
    3    <- getf
    3    -> fop_rwlock
    3      -> nfs4_rwlock
    3        -> nfs_rw_enter_sig
    3        <- nfs_rw_enter_sig
    3      <- nfs4_rwlock
    3    <- fop_rwlock
    3    -> fop_write
    3      -> nfs4_write
    3        -> nfs_rw_enter_sig
    3        <- nfs_rw_enter_sig
    3        -> uio_prefaultpages
    3        <- uio_prefaultpages
    3        -> writerp4
    3          -> vpm_data_copy
[...]
  3  <- nfs4_write
  3  <- fop_write
  3  -> fop_rwunlock
  3    -> nfs4_rwunlock
  3      -> nfs_rw_exit
  3        -> cv_broadcast
  3        <- cv_broadcast
  3      <- nfs_rw_exit
  3    <- nfs4_rwunlock
  3  <- fop_rwunlock
  3  -> releasef
  3    -> cv_broadcast
  3    <- cv_broadcast
  3  <- releasef
  3  <- write
  3  <= write
```

Most of the resulting output was truncated for space purposes (the total output was more than 400 lines). With the `flowindent` option, the output includes arrows and indentation based on function entry and return, allowing us to easily see function flow. The output spans from the entry point of the `write(2)` system call through to its return to the calling process. This information can be used in several ways, depending on your goal. As noted previously, it is extremely useful for understanding kernel internals, using the output in conjunction with the source code.

We can use the same script on Mac OS X:

```
macosx# ./ktrace.d read_nocancel
dtrace: script './ktrace.d' matched 18393 probes

CPU FUNCTION
  0  -> read_nocancel
  0    -> proc_fdlock_spin
  0    <- proc_fdlock_spin
  0    -> lck_mtx_lock_spin
  0    <- lck_mtx_lock_spin
  0    -> fp_lookup
  0    <- fp_lookup
```

```
  0    -> proc_fdunlock
  0    <- proc_fdunlock
  0    -> lck_mtx_unlock_darwin10
  0    <- lck_mtx_unlock_darwin10
  0    -> vfs_context_current
  0    <- vfs_context_current
  0    -> vfs_context_proc
  0      -> get_bsdthreadtask_info
  0      <- get_bsdthreadtask_info
[...]
  0      -> proc_fdunlock
  0      <- proc_fdunlock
  0      -> lck_mtx_unlock_darwin10
  0      <- lck_mtx_unlock_darwin10
  0    <- read_nocancel
  0  <= read_nocancel
```

And we can use it on FreeBSD:

```
freebsd# ./ktrace.d read
dtrace: script './ktrace.d' matched 37134 probes
CPU FUNCTION
  0  -> read
  0    -> kern_readv
  0      -> fget_read
  0        -> fget_unlocked
  0        <- fget_unlocked
  0      <- fget_read
  0      -> dofileread
  0        -> devfs_read_f
[...]
  0          <- random_read
  0          -> vfs_timestamp
  0          <- vfs_timestamp
  0          -> dev_relthread
  0            -> dev_lock
  0            <- dev_lock
  0            -> dev_unlock
  0            <- dev_unlock
  0          <- dev_relthread
  0        <- devfs_read_f
  0      <- dofileread
  0    <- kern_readv
  0  <- read
  0  <= read
```

For performance-related work with DTrace, this provides a clear view into what specific functions in the kernel may be candidates for timing. Try looking for higher-level functions that are suitable for tracing, rather than lower-level functions that may be more frequent and expensive to trace.

While `flowindent` is a convenient way to trace function flow, it isn't reliable: The output can become shuffled on multi-CPU systems, making function flow difficult to follow. One way to improve this is to include a time stamp in the output and to postsort based on that.

Extending the previous `ktrace.d` example from a Solaris system. If we want to take a closer look at writes, we can start by determining which file system type is the most frequent write target:

```
solaris# dtrace -n 'syscall::write:entry { @[fds[arg0].fi_fs] = count(); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

  tmpfs                                                                28
  specfs                                                               52
  lofs                                                                201
  fifofs                                                              278
  zfs                                                                 359
  nfs4                                                               3178
```

Most of the writes are to an NFSv4 mounted file system. We can modify `ktrace.d` to trace only the kernel flow for writes to NFS:

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent

syscall::write:entry
/fds[arg0].fi_fs == $$1/
{
        self->flag = 1;
}
fbt:::
/self->flag/
{
}
syscall::write:return
/self->flag/
{
        self->flag = 0;
        exit(0);
}
```

Script kwtrace.d

Note the `kwtrace.d` script has the write system call specified in the probe function field and includes a predicate so we execute the action in the clause only if the target file system matches what is specified on the command line:

```
solaris# ./kwtrace.d nfs4
dtrace: script './kwtrace.d' matched 68135 probes
CPU FUNCTION
  5  -> write
  5    -> getf
  5      -> set_active_fd
  5      <- set_active_fd
  5    <- getf
  5    -> fop_rwlock
```

```
  5      -> nfs4_rwlock
  5        -> nfs_rw_enter_sig
  5        <- nfs_rw_enter_sig
  5      <- nfs4_rwlock
  5    <- fop_rwlock
  5    -> fop_write
  5      -> nfs4_write
  5        -> nfs_rw_enter_sig
  5        <- nfs_rw_enter_sig
  5        -> uio_prefaultpages
  5        <- uio_prefaultpages
  5        -> writerp4
  5          -> vpm_data_copy    <--- kernel data copy entry point
  5            -> vpm_map_pages
[...]
  5              <- free_vpmap
  5            <- vpm_unmap_pages
  5          <- vpm_data_copy   <--- kernel data copy return
  5        <- writerp4
  5        -> vpm_sync_pages
  5        <- vpm_sync_pages
[...]
  5    -> releasef
  5      -> cv_broadcast
  5      <- cv_broadcast
  5    <- releasef
  5  <- write
  5  <- write
```

Having passed the string `nfs4` on the command line, `kwtrace.d` provides the kernel code path for the NFSv4 write. If we want to measure the time in a specific kernel function, we can now create a script based on this output. We'll take a look at the total time for a write system call to NFS and break out the kernel internal data copy component of the load by measuring the time spent in `vpm_data_copy()`, which we see in the `kwtrace.d` output. We can create a DTrace script that we can reuse specifically for chasing NFSv4 writes, measuring the kernel function specified on the command line.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall::write:entry
/fds[arg0].fi_fs == "nfs4"/
{
        self->st = timestamp;
}
fbt::$1:entry
/self->st/
{
        self->kst[probefunc] = timestamp;
}
fbt::$1:return
/self->kst[probefunc]/
{
        @ktime[probefunc] = sum(timestamp - self->kst[probefunc]);
```

continues

```
        self->kst[probefunc] = 0;
}
syscall::write:return
/self->st/
{
        @write_syscall_time = sum(timestamp - self->st);
        self->st = 0;
        exit(0);
}
END
{
        printa("Write syscall: %@d (nanoseconds)\n", @write_syscall_time);
        printa("Kernel function %s() time: %@d (nanoseconds)\n", @ktime);
}
```

Script writek.d

The `writek.d` script takes the name of the desired kernel function as a command-line argument, used in the probe function field of the fbt provider probes. A time stamp is captured both at the entry point of the write system call and the entry point of the specified kernel function and again at the return points to determine total time spent for the write and the kernel function, which will of course be a subset of the total time:

```
solaris# ./writek.d vpm_data_copy
Write syscall: 202027 (nanoseconds)
Kernel function vpm_data_copy() time: 20095 (nanoseconds)
```

The resulting output shows the write system call took about 200 microseconds, of which 20 microseconds was spent in `vpm_data_copy()`. Note again that, based on the kernel code flow observed earlier, we know that `vpm_data_copy()` calls other kernel functions. The measured time includes the called functions, often referred to as *inclusive* time in software profiling tools.

Kernel Memory Usage

In Chapter 3, System View, we briefly discussed tracking kernel memory allocation and consumption with DTrace, showing examples from memory allocators for the different operating systems. In this section, we will continue exploring kernel memory usage with DTrace.

An operating system kernel may support a variety of different kernel memory allocators, including page, zone, and slab. See the kernel texts listed at the start of this chapter for the complete reference of allocators available and their function. To get a sense of those currently in use, you can start by using the existing system tools to show kernel memory usage:

Solaris: `echo ::kmastat | mdb -k`

Mac OS X: `zprint`

FreeBSD: `vmstat -m`, `vmstat -z`

Each of these will list kernel memory zones with allocation sizes, showing where the most memory has been allocated.

DTrace can be used to watch allocations in flight by tracing the kernel functions performing them. This can also show which are the popular memory allocators, because each has their own interface functions. At a guess, these functions probably contain the word `alloc`:

```
solaris# dtrace -n 'fbt::*alloc*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::*alloc*:entry ' matched 703 probes
^C

  callbparams_alloc                                               1
  log_alloc                                                       1
  mi_copyout_alloc                                                1
  mi_tpi_trailer_alloc                                            1
  pt_ttys_alloc                                                   1
[...]
  kmem_slab_alloc                                              8137
  kmem_slab_alloc_impl                                         8137
  kmem_depot_alloc                                            21437
  nvp_buf_alloc                                               22259
  hment_alloc                                                 22450
  nv_mem_zalloc                                               28268
  nv_alloc_sys                                                31074
  zfs_acl_alloc                                               31908
  zfs_acl_node_alloc                                          31908
  kmem_alloc                                                  96147
  kmem_zalloc                                                112147
  kmem_cache_alloc                                           275312
```

This one-liner frequency counted kernel `alloc` functions on Solaris and found that the most frequent while tracing was `kmem_cache_alloc()`, called 275,312 times. This function is for the slab allocator.

Solaris, Mac OS X, and FreeBSD all have implementations of the slab allocator[2] for the allocation and management of reusable kernel objects, and they include a similar set of kernel functions prefixed with `kmem_`, such as `kmem_alloc()`, and so on. When using DTrace to instrument slab allocator allocation functions, it is important to note that they do not necessarily result in the allocation of physical memory. The design is based on object reuse, so a `kmem_alloc()` call may return the address of a previously freed kernel object that is already backed with physical memory. That said, it is still useful to understand which kernel subsystems are

2. This comes from *The Slab Allocator: An Object-Caching Kernel Memory Allocator* by Jeff Bonwick.

calling into the kernel allocation routines, as an indicator of which kernel caches are potentially growing.

For Solaris systems, we can instrument `kmem_cache_alloc()` and `kmem_cache_free()` to observe which object caches are most active.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt::kmem_cache_alloc:entry
{
        @alloc[args[0]->cache_name] = count();
}
fbt::kmem_cache_free:entry
{
        @free[args[0]->cache_name] = count();
}
tick-1sec
{
        printf("%-32s %-8s %-8s\n", "CACHE NAME", "ALLOCS", "FREES");
        printa("%-32s %-@8d %-@8d\n", @alloc, @free);
        trunc(@alloc); trunc(@free);
}
```

Script kmem_track.d

The `kmem_track.d` script simply counts entries into the allocate and free routine, aggregating on the name of the object cache in the kernel and generating output every second:

```
solaris# ./kmem_track.d
^C
CACHE NAME                        ALLOCS   FREES
[...]
streams_dblk_144                  623      623
kmem_alloc_256                    1914     12
kmem_alloc_128                    1917     1910
streams_mblk                      1946     1946
kmem_alloc_64                     1984     1975
streams_dblk_esb                  2037     2037
kmem_alloc_8                      2733     2730
zio_cache                         2890     2890
streams_dblk_208                  2977     2936
kmem_alloc_80                     3807     3807
kmem_alloc_32                     4836     4820
kmem_alloc_40                     5752     5745
streams_dblk_80                   5931     5864
kmem_alloc_16                     10540    10537
```

The output shows a very close balance in terms of allocations and frees for most of the object caches. For the `kmem_alloc_256` cache, we see significantly more allocations than frees, so we can take a closer look by using a predicate and instrumenting just the allocation function.

```
solaris# dtrace -n 'fbt::kmem_cache_alloc:entry /args[0]->cache_name ==
"kmem_alloc_256"/ { @[stack()] = count(); }'
dtrace: description 'fbt::kmem_cache_alloc:entry ' matched 1 probe
^C
[...]
              genunix`kmem_alloc+0x2c
              zfs`vdev_disk_io_start+0x25c
              zfs`zio_execute+0x74
              zfs`vdev_queue_io_done+0x84
              zfs`vdev_disk_io_done+0x4
              zfs`zio_execute+0x74
              genunix`taskq_thread+0x1a4
              unix`thread_start+0x4
             1663

              genunix`kmem_alloc+0x2c
              zfs`zil_itx_create+0x18
              zfs`zfs_log_write+0x100
              zfs`zfs_write+0x534
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
           245284
```

The output shows us that the most frequent kernel code path leading to cache allocations from the `kmem_alloc_256` cache are through the ZFS code path, with 245,284 occurrences of that stack frame during the sampling period vs. the next most frequent stack frame, which occurred only 1,663 times during the data collection (and was also the ZFS subsystem).

Another approach to examining Solaris kernel memory allocator activity is to instrument the `kmem_alloc()` function, tracking the size of the request and the kernel stack leading up to the call:

```
solaris# dtrace -n 'fbt::kmem_alloc:entry { @[arg0, stack()] = count(); }'
dtrace: description 'fbt::kmem_alloc:entry ' matched 1 probe
^C
[...]
               64 <--------------------------- size (arg0)
              zfs`zfs_range_lock+0xc
              zfs`zfs_write+0x160
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
            44668
               32 <--------------------------- size (arg0)
              zfs`dsl_dir_tempreserve_space+0x38
              zfs`dmu_tx_try_assign+0x228
              zfs`dmu_tx_assign+0xc
              zfs`zfs_write+0x314
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
            44696
               32 <--------------------------- size (arg0)
              zfs`zio_push_transform+0x8
              zfs`zio_create+0x110
```

continues

```
              zfs`zio_null+0x4c
              zfs`dmu_buf_hold_array_by_dnode+0xdc
              zfs`dmu_buf_hold_array+0x60
              zfs`dmu_write_uio+0x48
              zfs`zfs_write+0x40c
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
            44696
              232 <---------------------------- size (arg0)
              zfs`zil_itx_create+0x18
              zfs`zfs_log_write+0x100
              zfs`zfs_write+0x534
              genunix`fop_write+0x20
              genunix`write+0x268
              unix`syscall_trap+0xac
            44696
```

The output here shows again a kernel code path through ZFS, with the size of the `kmem_alloc()` request at the top of each stack frame.

The underlying mechanism for allocating physical memory in Solaris requires calling into the `segkmem` routines. We can instrument the memory allocator code in `segkmem` to observe physical memory allocations into the kernel address space.

```
solaris# dtrace -n 'fbt::segkmem_xalloc:entry { @segkmem[args[0]->vm_name,
arg2, stack()] = count(); }'
dtrace: description 'fbt::segkmem_xalloc:entry ' matched 1 probe
^C
[...]
  heap                                                          12288
              unix`segkmem_alloc_io_4G+0x26
              genunix`vmem_xalloc+0x315
              genunix`vmem_alloc+0x155
              unix`kalloca+0x160
              unix`i_ddi_mem_alloc+0xd6
              rootnex`rootnex_setup_copybuf+0xe4
              rootnex`rootnex_bind_slowpath+0x2dd
              rootnex`rootnex_coredma_bindhdl+0x16c
              rootnex`rootnex_dma_bindhdl+0x1a
              genunix`ddi_dma_buf_bind_handle+0xb0
              sata`sata_dma_buf_setup+0x4b9
              sata`sata_scsi_init_pkt+0x1f5
              scsi`scsi_init_pkt+0x44
              sd`sd_setup_rw_pkt+0xe5
              sd`sd_initpkt_for_buf+0xa3
              sd`sd_start_cmds+0xa5
              sd`sd_return_command+0xd7
              sd`sdintr+0x187
              sata`sata_txlt_rw_completion+0x145
              nv_sata`nv_complete_io+0x95
               90
  heap <----- vmem name                                          8192 <---- size
              unix`segkmem_alloc_io_4G+0x26
              genunix`vmem_xalloc+0x315
              genunix`vmem_alloc+0x155
              unix`kalloca+0x160
              unix`i_ddi_mem_alloc+0xd6
              rootnex`rootnex_setup_copybuf+0xe4
              rootnex`rootnex_bind_slowpath+0x2dd
              rootnex`rootnex_coredma_bindhdl+0x16c
```

```
                rootnex`rootnex_dma_bindhdl+0x1a
                genunix`ddi_dma_buf_bind_handle+0xb0
                sata`sata_dma_buf_setup+0x4b9
                sata`sata_scsi_init_pkt+0x1f5
                scsi`scsi_init_pkt+0x44
                sd`sd_setup_rw_pkt+0xe5
                sd`sd_initpkt_for_buf+0xa3
                sd`sd_start_cmds+0xa5
                sd`sd_return_command+0xd7
                sd`sdintr+0x187
                sata`sata_txlt_rw_completion+0x145
                nv_sata`nv_complete_io+0x95
                142
```

Here we see memory allocation for the kernel through the SCSI Disk (sd driver) into the kernel's vmem heap segment (memory for kernel object caches are allocated out of the kernel heap[3]).

Using the same basic methodology used in the `kmem_track.d` script, we can track allocations and frees in the `segkmem` code:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt::segkmem_xalloc:entry
{
        @segkmem_alloc[args[0]->vm_name, arg2] = count();
}
fbt::segkmem_free_vn:entry
{
         @segkmem_free[args[0]->vm_name, arg2] = count();
}
END
{
         printf("%-16s %-8s %-8s %-8s\n", "VMEM NAME", "SIZE", "ALLOCS", "FREES");
         printa("%-16s %-8d %-@8d %-@8d\n", @segkmem_alloc, @segkmem_free);
}
```

Script segkmem.d

```
solaris# ./segkmem.d
^C
VMEM NAME        SIZE     ALLOCS   FREES
heap             73728    2        0
heap             1933312  6        4
heap             278528   24       24
heap             16384    49       49
```

As was the case with the kmem layer, we can see a pretty even number of allocations and frees. The key point here is that, when tracking kernel memory, we need to examine both to determine actual physical memory growth.

3. This comes from *Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources* by Jeff Bonwick and Jonathan Adams.

On Mac OS X, `kernel_memory_allocate()` is a master function for kernel memory allocations (but not the only one). The following DTrace command line shows the source of kernel memory requests by instrumenting the entry point of this function and aggregating on the process name, the size of the allocation (`arg2`), and the kernel stack:

```
macosx# dtrace -n 'fbt::kernel_memory_allocate:entry { @[execname, arg2, stack()] =
 count(); }'
dtrace: description 'fbt::kernel_memory_allocate:entry ' matched 1 probe
^C
[...]
  mds                                                           65541
              mach_kernel`kmem_alloc+0x38
              mach_kernel`kalloc_canblock+0x76
              mach_kernel`OSMalloc+0x60
              0x5a5bce0a
              0x5a5be95b
              0x5a5befcc
              mach_kernel`decmpfs_hides_rsrc+0x5f3
              mach_kernel`decmpfs_pagein_compressed+0x1b6
              mach_kernel`hfs_vnop_pagein+0x64
              mach_kernel`VNOP_PAGEIN+0x9e
              mach_kernel`vnode_pagein+0x30b
              mach_kernel`vnode_pager_cluster_read+0x5c
              mach_kernel`vnode_pager_data_request+0x8a
              mach_kernel`vm_fault_page+0xcaa
              mach_kernel`vm_fault+0xd2d
              mach_kernel`user_trap+0x29f
              mach_kernel`lo_alltraps+0x12a
              392
  WindowServer                                                   8736
              mach_kernel`kmem_alloc+0x38
              mach_kernel`kalloc_canblock+0x76
              mach_kernel`kalloc+0x19
              mach_kernel`IOMalloc+0x12
              0x5afd6d2e
              0x5afd9241
              mach_kernel`shim_io_connect_method_structureI_structureO+0x15e
              mach_kernel`IOUserClient::externalMethod+0x3c0
              mach_kernel`is_io_connect_method+0x1d3
              mach_kernel`iokit_server_routine+0x123d
              mach_kernel`ipc_kobject_server+0xf4
              mach_kernel`ipc_kmsg_send+0x6f
              mach_kernel`mach_msg_overwrite_trap+0x112
              mach_kernel`thread_setuserstack+0x195
              mach_kernel`lo64_mach_scall+0x4d
              829
  WindowServer                                                   8736
              mach_kernel`kmem_alloc+0x38
              mach_kernel`kalloc_canblock+0x76
              mach_kernel`kalloc+0x19
              mach_kernel`IOMalloc+0x12
              0x5afd694f
              0x5afd92f5
              mach_kernel`shim_io_connect_method_structureI_structureO+0x15e
              mach_kernel`IOUserClient::externalMethod+0x3c0
              mach_kernel`is_io_connect_method+0x1d3
              mach_kernel`iokit_server_routine+0x123d
              mach_kernel`ipc_kobject_server+0xf4
              mach_kernel`ipc_kmsg_send+0x6f
              mach_kernel`mach_msg_overwrite_trap+0x112
              mach_kernel`thread_setuserstack+0x195
```

```
                mach_kernel`lo64_mach_scall+0x4d
                829
```

Looking at the last item in the output, we see the most frequent kernel stack frame occurred 829 times during the tracing period, the process on-cpu was the Mac OS X WindowServer, and the size passed to the `kernel_memory_allocate()` function was 8736. In this sample, the last two entries are actually very similar: the count value, size value, and process name. The stack frames are also very similar, both showing the path to kernel memory allocation originating with a system call (`mach_kernel`lo64_mach_scall+0x4d`) and moving up through the Mac OS X Interprocess Communication (IPC) and IO Kit path.

Alternatively, using the `quantize` aggregating function and `execname`, a more summarized view of kernel memory allocations is generated:

```
macosx# dtrace -n 'fbt::kernel_memory_allocate:entry { @[execname] =
quantize(arg2); }'
dtrace: description 'fbt::kernel_memory_allocate:entry ' matched 1 probe
^C
  Kindle for Mac
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@                                 6
            8192 |@@@@@@@@@@@@@@@@@@@@@                    17
           16384 |                                         0
           32768 |@@@                                      2
           65536 |@@@                                      2
          131072 |@@@@@                                    4
          262144 |                                         0
          524288 |                                         0
         1048576 |@                                        1
         2097152 |                                         0

  AppleSpell
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@                     40
           16384 |@                                        1
           32768 |@                                        1
           65536 |@@@@@@@@@@@@@@@@@@@                      37
          131072 |                                         0

  NoteBook
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@                                      21
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             188
           16384 |                                         3
           32768 |@@@@                                     26
           65536 |@@@                                      20
          131072 |@                                        8
          262144 |                                         0

  WindowServer
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |                                         5
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1120
           16384 |                                         0
```

The output shows the process name and a graph of the size of the requested allocations. We can see the WindowServer generated mostly 8KB to 16KB size requests, and an application called *NoteBook* also fell in the 8KB to 16KB range, with a few allocations in the 128KB to 256KB range.

Instrumenting the Mac OS X kmem alloc and free calls, along with the sizes, provides another point of observability into kernel memory activity:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

fbt::kmem_alloc:entry
{
   @alloc[arg2] = count();
}
fbt::kmem_free:entry
{
   @free[arg2] = count();
}
END
{
   printf("%-16s %-8s %-8s\n", "SIZE", "ALLOCS", "FREES");
   printa("%-16d %-@8d %-@8d\n", @alloc, @free);
}
```

Script kmem_osx.d

```
macosx# ./kmem_osx.d
^C
SIZE             ALLOCS   FREES
18672            0        1
8330             0        2
9935             0        2
110313           0        2
8334             1        0
9939             1        0
110317           1        0
60               1        1
80               1        1
11680            1        1
15176            1        1
30352            1        1
16384            1        2
8736             2        2
11660            2        2
15244            2        2
1048576          2        2
11648            9        9
4096             92       198
```

In FreeBSD, a similar set of scripts tracking `kmem_alloc()` and `kmem_free()`, aggregating on kernel stack frames and `execname` (process names), can be used to understand kernel memory usage.

There are more kernel allocators than shown here, all of which can be explored with DTrace. For example, try tracing the `kalloc()` and `zalloc()` functions on

Mac OS X and the `malloc()` function on FreeBSD. Sizes and stack traces can be explored in similar ways as shown here for the slab allocator.

Anonymous Tracing

This feature allows DTrace to be enabled when there is no user-land consumer (such as `dtrace(1M)`) running. This is particularly powerful for kernel analysis, because it can be used to investigate device driver issues during boot time, before processes are running.

To introduce this with a well-known topic rather than what would likely be an unfamiliar device driver, a nonkernel example is demonstrated next: analysis of boot processes.

The `-A` option to `dtrace(1M)` saves the D program specified (either with `-n` or `-s` for a script) into the `/kernel/drv/dtrace.conf` file to be read at boot time (which happens very early, before most other drivers):

```
solaris# dtrace -A -qn 'proc:::exec-success { printf("%-10d %-6d %-6d %s\n",
timestamp, ppid, pid, curpsinfo->pr_psargs); }'
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
```

This D program traces process execution, printing a time stamp and the parent process ID, process ID, and process argument list. It won't be enabled until the server is rebooted. After the reboot, the DTrace data can be collected using `-a`:

```
solaris# dtrace -a > boot.out
solaris# sort -n boot.out
215913067635 0      1      /sbin/init
216070909233 1      6      INITSH -c exec /sbin/autopush -f /etc/iu.ap
216271075785 1      6      /sbin/autopush -f /etc/iu.ap
216672994689 1      7      INITSH -c exec /sbin/soconfig -f /etc/sock2path
216687058247 1      7      /sbin/soconfig -f /etc/sock2path
[...truncated...]
302497182805 1742   1776   /lib/svc/bin/lsvcrun /etc/rc2.d/S89PRESERVE start
302503684531 1776   1777   /bin/sh /etc/rc2.d/S89PRESERVE start
302525623084 1742   1778   /lib/svc/bin/lsvcrun /etc/rc2.d/S98deallocate start
302531891396 1778   1779   /bin/sh /etc/rc2.d/S98deallocate start
302552852919 1779   1780   /usr/sbin/auditconfig -getcond
302558707682 1779   1781   /usr/sbin/deallocate -Is
302799248939 1742   1783   /usr/sbin/devfsadm -S
303534313252 76     1784   /sbin/sh -c exec /sbin/rc3
303548242737 76     1784   /sbin/sh /sbin/rc3
303557236068 1784   1785   /usr/bin/who -r
303561859363 1784   1787   /usr/bin/uname -a
303565852737 1784   1788   /sbin/netstrategy
303682501117 76     1786   /sbin/sh -c exec /lib/svc/method/svc-boot-config
303690686594 76     1786   /sbin/sh /lib/svc/method/svc-boot-config
303704109035 1786   1791   /usr/sbin/uadmin 23 1
```

continues

```
303816877333 76     1790   /sbin/sh -c exec /lib/svc/method/svc-intrd
303824547597 76     1790   /sbin/sh /lib/svc/method/svc-intrd
[...truncated...]
```

The output of `-a` will be shuffled on multi-CPU servers and so was sorted on the included time stamp. The first process to execute during boot was `/sbin/init` (as would be expected).

A section of output was truncated so that later boot execution is also shown, starting with the (legacy) `/etc/rc2.d` scripts (extra lines were also truncated, caused by `"\n"` characters in `curpsinfo->pr_psargs`). There is enough data in this output to investigate boot latency by process and to follow the parent process IDs to the origin, such as a start script. The script could be enhanced to show both the process start and end times and other sources of latency including disk I/O. (Such a project was undertaken during development of Solaris 10; DTrace was used to track down boot latency issues that were then resolved.[4])

For anonymous tracing to work in debugging kernel device driver initialization, the `dtrace` module needs to be loaded before the device driver. It is in fact loaded very early in the boot process; the `modinfo(1M)` command shows the load order:

```
solaris# modinfo
 Id         Loadaddr   Size Info Rev Module Name
  0 fffffffffb800000 1d80c2   -   0  unix ()
  1 fffffffffb957360 2ef9d0   -   0  genunix ()
  3 fffffffffbbe0000   5e20   1   1  specfs (filesystem for specfs)
  4 fffffffffbbe5d80   46a8   3   1  fifofs (filesystem for fifo)
  5 fffffffff7c84000  1cd08  20   1  dtrace (Dynamic Tracing)
  6 fffffffffbbea370   5c80  16   1  devfs (devices filesystem)
  7 fffffffff77fb000  118c0  17   1  dev (/dev filesystem)
  8 fffffffffbbefda8   6570   -   1  dls (Data-Link Services)
  9 fffffffff780d000  322c8   -   1  mac (MAC Services)
 10 fffffffff783f000  21a98   5   1  procfs (filesystem for proc)
 12 fffffffffbbf6020   4148   1   1  TS (time sharing sched class)
 13 fffffffff780c2c0    9e8   -   1  TS_DPTBL (Time sharing dispatch table)
 14 fffffffff7861000   a060   -   1  pci_autoconfig (PCI BIOS interface)
 15 fffffffff786b000  61e10   -   1  acpica (ACPI interpreter)
 16 fffffffff78cc000  18940   -   1  pcie (PCI Express Framework Module)
[...]
```

The `dtrace` module is near the top, with ID 5.

One-Liners

The following one-liners can be used to profiling the kernel and tracking system events of interest.

---

4. See *http://blogs.sun.com/dp/entry/more_on_bootchart_for_solaris* by Dan Price and *http://blogs.sun.com/eschrock/entry/boot_chart_results* by Eric Schrock.

syscall Provider

Count system calls by type:

```
dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
```

profile Provider

Kernel stack trace profile at 1001 Hertz:

```
dtrace -n 'profile-1001 { @[stack()] = count(); }'
```

Kernel stack trace profile at 1001 Hertz, top five stack frame functions per stack:

```
dtrace -n 'profile-1001 { @[stack(5)] = count(); }'
```

Kernel stack trace profile at 1001 Hertz, top 20 stacks:

```
dtrace -n 'profile-1001 { @[stack()] = count(); } END { trunc(@, 20); }'
```

Kernel function name profile at 1001 Hertz:

```
dtrace -n 'profile-1001 { @[func(arg0)] = count(); }'
```

Kernel module name profile at 1001 Hertz:

```
dtrace -n 'profile-1001 { @[mod(arg0)] = count(); }'
```

Kernel thread name profile at 1001 Hertz (FreeBSD):

```
dtrace -n 'profile-1001 { @[stringof(curthread->td_name)] = count(); }'
```

sched Provider

Thread off-cpu stack trace count:

```
dtrace -n 'sched:::off-cpu { @[stack()] = count(); }'
```

Stack size for processes (Solaris):

```
dtrace -n 'sched:::on-cpu { @[execname] = max(curthread->t_procp->p_stksize); }'
```

vminfo provider

Pages paged in by process name:

```
dtrace -n 'vminfo:::pgpgin { @pg[execname] = sum(arg0); }'
```

Minor faults by process name:

```
dtrace -n 'vminfo:::as_fault { @mem[execname] = sum(arg0); }'
```

sysinfo Provider

CPU cross calls by process name:

```
dtrace -n 'sysinfo:::xcalls { @[execname] = count(); }'
```

CPU cross calls by kernel stack trace:

```
dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); }'
```

lockstat Provider

Adaptive lock block time totals (ns) by process name:

```
dtrace -n 'lockstat:::adaptive-block { @time[execname] = sum(arg1); }'
```

Adaptive lock block time distribution (ns) by process name:

```
dtrace -n 'lockstat:::adaptive-block { @time[execname] = quantize(arg1); }'
```

Adaptive lock block time totals (ns) by kernel stack trace:

```
dtrace -qn 'lockstat:::adaptive-block { @[stack(5), "^^^ total ns:"] = sum(arg1); }'
```

Adaptive lock block time totals (ns) by lock name (if symbol data is present):

```
dtrace -qn 'lockstat:::adaptive-block { @[arg0] = sum(arg1); } END { printa("%40a
%@16d ns\n", @); }'
```

Adaptive lock block time totals (ns) by calling function:

```
dtrace -qn 'lockstat:::adaptive-block { @[caller] = sum(arg1); } END { printa("%40a
%@16d ns\n", @); }'
```

sdt Provider

Count interrupts by CPU:

```
dtrace -n 'sdt:::interrupt-start { @num[cpu] = count(); }'
```

fbt Provider

The fbt provider instruments a particular operating system and version and hence is considered unstable. This means that the following one-liners may require modifications to match the software version you are running. This is only a sample of the thousands of possible fbt provider–based one-liners for examining kernel operation.

Kernel function call counts:

```
dtrace -n 'fbt:::entry { @[probefunc] = count(); }'
```

Kernel function call counts by module:

```
dtrace -n 'fbt:::entry { @[probemod] = count(); }'
```

Kernel function call counts for module `zfs` by module:

```
dtrace -n 'fbt:zfs::entry { @[probefunc] = count(); }'
```

Kernel function call counts for functions beginning with `hfs_` by module:

```
dtrace -n 'fbt::hfs_*:entry { @[probefunc] = count(); }'
```

Kernel stack backtrace counts for calls to function `arc_read()` (for example):

```
dtrace -n 'fbt::arc_read:entry { @[stack()] = count(); }'
```

Count kernel `alloc` functions to investigate kernel memory allocation:

```
dtrace -n 'fbt::*alloc*:entry { @[probefunc] = count(); }'
```

Kernel kmem cache allocations by cache name (Solaris):

```
dtrace -n 'fbt::kmem_cache_alloc:entry { @[args[0]->cache_name] = count(); }'
```

Kernel `kernel_memory_allocate()` calls by stack trace (Mac OS X):

```
dtrace -n 'fbt::kernel_memory_allocate:entry { @[stack()] = count(); }'
```

Kernel `malloc()` calls by malloc type and size distribution (FreeBSD):

```
dtrace -n 'fbt::malloc:entry { @[stringof(args[1]->ks_shortdesc)] = quantize(arg1); }'
```

Show who is calling `delay()` and for how many clock ticks (snoozers):

```
dtrace -n 'fbt::delay:entry { @[stack()] = quantize(arg0); }'
```

Show who is calling `pause()`, why, and for how long in ticks (FreeBSD):

```
dtrace -n 'fbt::pause:entry { @[stack(), stringof(arg0)] = quantize(arg1); }'
```

cpc Provider

These cpc provider one-liners are dependent on the availability of both the cpc provider and the event probes (for Solaris, see `cpustat(1M)` to see what events are available on your system). The following overflow counts (200,000; 50,000; and 10,000) have been picked to balance between the rate of CPC events and fired DTrace probes.

Kernel-mode instructions by thread address:

```
dtrace -n 'cpc:::PAPI_tot_ins-kernel-200000 { @[(uint64_t)curthread] = count(); }'
```

Kernel-mode instructions by function name:

```
dtrace -n 'cpc:::PAPI_tot_ins-kernel-200000 { @[func(arg0)] = count(); }'
```

Kernel-mode instructions by module name:

```
dtrace -n 'cpc:::PAPI_tot_ins-kernel-200000 { @[mod(arg0)] = count(); }'
```

Kernel-mode CPU cycles by function name:

```
dtrace -n 'cpc:::PAPI_tot_cyc-kernel-200000 { @[func(arg0)] = count(); }'
```

Kernel-mode level-one cache misses by function name:

```
dtrace -n 'cpc:::PAPI_l1_tcm-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode level-one instruction cache misses by function name:

```
dtrace -n 'cpc:::PAPI_l1_icm-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode level-one data cache misses by function name:

```
dtrace -n 'cpc:::PAPI_l1_dcm-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode level 2 cache misses by function name:

```
dtrace -n 'cpc:::PAPI_l2_tcm-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode level 3 cache misses by function name:

```
dtrace -n 'cpc:::PAPI_l3_tcm-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode conditional branch misprediction by function name:

```
dtrace -n 'cpc:::PAPI_br_msp-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode resource stall cycles by function name:

```
dtrace -n 'cpc:::PAPI_res_stl-kernel-50000 { @[func(arg0)] = count(); }'
```

Kernel-mode floating-point operations by function name:

```
dtrace -n 'cpc:::PAPI_fp_ops-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode TLB misses by function name:

```
dtrace -n 'cpc:::PAPI_tlb_tl-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode instruction TLB misses by function name:

```
dtrace -n 'cpc:::PAPI_tlb_im-kernel-10000 { @[func(arg0)] = count(); }'
```

Kernel-mode data TLB misses by function name:

```
dtrace -n 'cpc:::PAPI_tlb_dm-kernel-10000 { @[func(arg0)] = count(); }'
```

One-Liner Selected Examples

Here we show examples of some of the one-liners from the previous section.

Count System Calls by Type

Although this is a simple one-liner, tracing system calls should not be overlooked when approaching the kernel. System calls are a main input to the kernel (the other being hardware interrupts), and checking what the kernel is being asked to do provides important context for what the kernel is actually doing. Here just the syscall types are counted by aggregating on the function name:

```
solaris# dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 233 probes
^C

  exece                                                              1
  fork1                                                              1
[...output truncated...]
  fchdir                                                          1367
  getdents64                                                      1371
  fstat64                                                         1376
  pathconf                                                        5359
  lstat64                                                         5388
  gtime                                                           5496
  pollsys                                                         5593
  ioctl                                                           5636
  acl                                                            10718
```

During tracing, a `find(1)` command was searching the file system while printing metadata (`find . -ls`). The most common syscall type was `acl()`, because `find` retrieved file metadata. From this, we would expect that the most frequently accessed kernel functions would be from the file system as it retrieved access control list (ACL) information for files; this may include performing device I/O to read the information from storage devices.

Kernel Stack Trace Profile at 1001 Hertz

This is one of the most useful DTrace one-liners, providing a quick look at why kernel code is on-CPU. The rate used is not that important; we chose 1001 Hertz to avoid lockstep sampling with events that might be running every millisecond.

```
solaris# dtrace -n 'profile-1001 { @[stack()] = count(); }'
dtrace: description 'profile-1001 ' matched 1 probe
^C
[...output truncated...]
              unix`do_splx+0x80
              unix`xc_common+0x231                    <--- CPU cross call
              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x193
              unix`hat_unload+0x41
              unix`segkmem_free_vn+0x6f
              unix`segkmem_free+0x27
              genunix`vmem_xfree+0x104
              genunix`vmem_free+0x29
              unix`kfreea+0x54
              unix`i_ddi_mem_free+0x5d
              rootnex`rootnex_teardown_copybuf+0x24
              rootnex`rootnex_coredma_unbindhdl+0xbd
              rootnex`rootnex_dma_unbindhdl+0x2e
              genunix`ddi_dma_unbind_handle+0x41
              sata`sata_common_free_dma_rsrcs+0x72
              sata`sata_scsi_destroy_pkt+0x2c
             1053

             1121                                    <--- User-land (no kernel stack)

              unix`do_copy_fault_nta+0x30            <--- Memory I/O
              genunix`uiomove+0xc6
              zfs`dmu_read_uio+0xa8
              zfs`zfs_read+0x19a
              genunix`fop_read+0xa7
              nfssrv`rfs3_read+0x3a1
              nfssrv`common_dispatch+0x384
              nfssrv`rfs_dispatch+0x2d
              rpcmod`svc_getreq+0x19c
              rpcmod`svc_run+0x16e
              rpcmod`svc_do_run+0x81
              nfs`nfssys+0x765
              unix`sys_syscall32+0xff
             3133

              zfs`fletcher_4_native+0x71
              zfs`zio_checksum_error+0x2d4           <--- Code path (ZFS checksum)
              zfs`zio_checksum_verify+0x3e
              zfs`zio_execute+0x89
              genunix`taskq_thread+0x1b7
              unix`thread_start+0x8
             3425

              unix`mach_cpu_idle+0x6                 <--- Idle
              unix`cpu_idle+0xaf
              unix`cpu_idle_adaptive+0x19
              unix`idle+0x114
              unix`thread_start+0x8
            26884
```

The output includes the kernel stack backtrace followed by a count for the number of times it was sampled on-CPU. The stack traces shown earlier have been identified, with the most common (listed last) being the idle loop. A user-land application thread was on-CPU for 1121 of the samples, generating no kernel stack trace (to see its stack trace, aggregate on `ustack()` instead).

Kernel Module Name Profile at 1001 Hertz

This one-liner samples the module name that is on-CPU (and currently doesn't work on Mac OS X, as mentioned earlier):

```
solaris# dtrace -n 'profile-1001 { @[mod(arg0)] = count(); }'
dtrace: description 'profile-1001 ' matched 1 probe
^C

  sd                                                                1
  mac                                                               2
  TS                                                                2
  ip                                                                4
  c2audit                                                          10
  genunix                                                         247
  0x0                                                             656
  zfs                                                             848
  unix                                                           4348
```

The `0x0` function is for user-land code. The hottest module was `unix`, which provides common functions for other modules. Although `unix` functions were hot on-CPU, they may be requested by other modules; using the previous one-liner will explain:

```
              unix`tsc_read+0x5
              genunix`gethrtime+0xd
              unix`pc_gethrestime+0x49
              genunix`gethrestime+0x19
              zfs`zfs_time_stamper_locked+0x2e
              zfs`zfs_time_stamper+0x40
              zfs`zfs_read+0x20c
              genunix`fop_read+0x6b
              genunix`read+0x2b8
              genunix`read32+0x22
              unix`_sys_sysenter_post_swapgs+0x14b
              526
```

The `tsc_read()` function from the `unix` module was called from a code path that includes ZFS.

Kernel Thread Name Profile at 1001 Hertz (FreeBSD):

The FreeBSD thread structure (/usr/src/sys/sys/proc.h) contains the name of the thread. Profiling the on-CPU thread name at 1001 Hertz yields the following:

```
freebsd# dtrace -n 'profile-1001 { @[stringof(curthread->td_name)] = count(); }'
dtrace: description 'profile-1001 ' matched 1 probe
dtrace: aggregation size lowered to 3m
^C

  sh                                                                3
  em0 taskq                                                         9
  swi4: clock                                                      19
  sshd                                                            108
  idle: cpu0                                                     1947
```

The output includes user-land threads, sh and sshd, as well as kernel threads including idle: cpu0. Having a human-readable name for a kernel thread is particularly handy for debugging with DTrace, rather than trying to determine a thread's function from what may be a cryptic stack trace alone.

CPU Cross Calls by Kernel Stack Trace

Although CPU cross calls are lightweight events, many thousands per second can cause performance problems because they frequently interrupt the operation of other CPUs. When excessive cross calls are identified (for example, the xcal field from mpstat(1M)), DTrace can be used to identify the reason for the cross calls:

```
solaris# dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); }'
dtrace: description 'sysinfo:::xcalls ' matched 2 probes
^C

              unix`xc_call+0x46
              unix`hat_tlb_inval+0x283
              unix`x86pte_inval+0xaa
              unix`hat_pte_unmap+0xfd
              unix`hat_unload_callback+0x23e
              unix`hat_unload+0x41
              genunix`segkp_release_internal+0xb5
              genunix`segkp_release+0xbd
              genunix`schedctl_freepage+0x33
              genunix`schedctl_proc_cleanup+0x5c
              genunix`proc_exit+0x1a6
              genunix`exit+0x15
              genunix`rexit+0x1c
              unix`sys_syscall32+0xff
                7
```

In this example, the cross calls were because of processes exiting and their memory address spaces being cleaned up by the hardware address translation (HAT) layer.

Kernel Function Call Counts for Functions Beginning with hfs_ by Module

Tracing HFS+ calls on Mac OS X while a file system archive operation is performed:

```
macosx# dtrace -n 'fbt::hfs_*:entry { @[probefunc] = count(); }'
dtrace: description 'fbt::hfs_*:entry ' matched 47 probes
^C

  hfs_vnop_write                                                    1
  hfs_generate_volume_notifications                                 2
  hfs_getinoquota                                                   2
  hfs_vnop_ioctl                                                    2
  hfs_chkdq                                                         3
  hfs_vnop_pagein                                                   3
  hfs_vnop_bwrite                                                   6
  hfs_vnop_blktooff                                                82
  hfs_swap_BTNode                                                  99
  hfs_hides_rsrc                                                  418
  hfs_vnop_blockmap                                               653
  hfs_vnop_strategy                                               659
  hfs_uncompressed_size_of_compressed_file                        936
  hfs_vnop_read                                                  1645
  hfs_hides_xattr                                                3691
  hfs_file_is_compressed                                         7667
```

The most frequently called function while tracing was `hfs_file_is_compressed()`.

Kernel Stack Backtrace Counts for Calls to Function foo()

The previous one-liner identified the function `hfs_file_is_compressed()` as frequently called; by tracing it and frequency counting the kernel stack trace, we can determine the reason it's being called:

```
macosx# dtrace -n 'fbt::hfs_file_is_compressed:entry { @[stack()] = count(); }'
dtrace: description 'fbt::hfs_file_is_compressed:entry ' matched 1 probe
^C
[...output truncated...]
              mach_kernel`hfs_uncompressed_size_of_compressed_file+0x197
              mach_kernel`VNOP_GETATTR+0x65
              mach_kernel`vnode_getattr+0x84
              mach_kernel`vn_stat_noauth+0xa1
              mach_kernel`pathconf+0x1a5
              mach_kernel`pathconf+0x556
              mach_kernel`lstat64+0x48
              mach_kernel`unix_syscall64+0x269
              mach_kernel`lo64_unix_scall+0x4d
             2100
```

The function was called during `vnode_getattr()`, presumably to fetch file attributes.

Kernel-Mode Instructions by Function Name

This one-liner uses the cpc provider to profile instructions by function, counting on every 200,000th instruction:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-kernel-200000 { @[func(arg0)] = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-kernel-200000 ' matched 1 probe
^C

  mac`mac_client_vid                                                  1
  mac`mac_stat_get                                                    1
  pcplusmp`apic_set_idlecpu                                           1
[...]
  genunix`avl_find                                                  342
  unix`x86pte_get                                                   350
  unix`x86pte_mapin                                                 377
  unix`htable_lookup                                                386
  unix`bzero                                                        504
  genunix`fsflush_do_pages                                          601
  unix`tsc_read                                                    1226
  unix`default_lock_delay                                          2416
  unix`mutex_enter                                                 2843
  unix`do_copy_fault_nta                                           5039
  unix`mutex_delay_default                                        11182
  zfs`fletcher_4_native                                           16918
  zfs`vdev_raidz_generate_parity_pq                               29703
```

While profiling, the zfs function `vdev_raidz_generate_parity_pq()` has executed the most instructions, based on the cpc profile used. The system has a ZFS file system with a write workload, and the one-liner has identified that the bulk of kernel instructions are spent calculating RAID-Z parity.

Kernel-Mode Instructions by Module Name

This one-liner counts instructions by kernel module (which currently works best on Solaris—see the earlier comments about module name availability in the "fbt Provider" section). The count is incremented on every 200,000th instruction:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-kernel-200000 { @[mod(arg0)] = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-kernel-200000 ' matched 1 probe
^C

  mm                                                                  1
  TS                                                                  1
  pcplusmp                                                            2
  specfs                                                              2
  SDC                                                                 5
  0x0                                                                 9
  kcf                                                                16
  sha1                                                               41
  scsi                                                               69
  sha2                                                               69
  scsi_vhci                                                          70
  sd                                                                181
```

```
  rootnex                                                           408
  mpt                                                               421
  dtrace                                                            454
  genunix                                                          2062
  unix                                                            19688
  zfs                                                             35620
```

According to this cpc profile, the zfs module was executing the most instructions.

Kernel-Mode Level-One Data Cache Misses by Function Name

This one-liner counts the current kernel function on every 10,000 level-one data cache miss for each CPU:

```
solaris# dtrace -n 'cpc:::PAPI_l1_dcm-kernel-10000 { @[func(arg0)] = count(); }'
dtrace: description 'cpc:::PAPI_l1_dcm-kernel-10000 ' matched 1 probe
^C

  rootnex`rootnex_teardown_copybuf                                    1
  rootnex`immu_map_sgl                                                1
[...]
  zfs`fletcher_4_native                                             142
  genunix`fsflush_do_pages                                          193
  unix`mutex_enter                                                  489
  unix`bzero                                                        507
  unix`tsc_read                                                     552
  unix`0xfffffffffb857cba                                           761
  zfs`vdev_raidz_generate_parity_pq                                 786
  unix`do_copy_fault_nta                                          20248
```

The function causing the most data cache misses was `do_copy_fault_nta()`, which is the kernel function to copy data (nontemporal access).

Kernel-Mode Level-One Instruction Cache Misses by Function Name

This one-liner counts the current kernel function on every 10,000 level-one instruction cache miss for each CPU:

```
solaris# dtrace -n 'cpc:::PAPI_l1_icm-kernel-10000 { @[func(arg0)] = count(); }'
dtrace: description 'cpc:::PAPI_l1_icm-kernel-10000 ' matched 1 probe
^C
^C

  pcplusmp`apic_send_ipi                                              1
  pcplusmp`apic_redistribute_compute                                  1
[...]
  unix`mutex_exit                                                    28
  scsi_vhci`vhci_bind_transport                                      31
  unix`rw_exit                                                       31
  sd`sd_return_command                                               36
  unix`bzero                                                         39
  unix`tsc_read                                                      40
  genunix`kmem_cache_alloc                                           50
  unix`mutex_enter                                                  242
```

The most instruction cache misses were from the `mutex_enter()` function. We found this surprising, thinking that such a frequently called function would remain in cache. The explanation may be that this function is being flushed from the cache on thread context switch, which happens frequently because of mutex blocks, followed by cache misses when the lock is acquired and the thread continues to execute. (We aren't sure—we just discovered this.)

Scripts

Table 12-3 summarizes the scripts that follow and the providers they use. The fbt and sdt scripts instrument a particular operating system kernel version (these scripts are for Solaris Nevada, circa June 2010). See the "fbt Provider" section earlier in this chapter for an explanation of the fbt provider interface. The last three scripts are from the DTraceToolkit (see Chapter 13, Tools).

intrstat

`intrstat(1M)` shows device interrupt statistics, including time spent servicing interrupts by device. This is a Solaris binary DTrace consumer, shipped under `/usr/sbin/intrstat`.

The inclusion of this DTrace-based tool helps complete system observability for CPU utilization. CPUs can be busy for a number of reasons, and tools such as `prstat(1M)` or `top(1)` only properly identify processes (PIDs) that are consuming CPU (because of hot user-land code and syscalls). `intrstat(1M)` on Solaris identifies device drivers that are consuming CPU because of interrupts. (You can

Table 12-3 Kernel Script Summary

Script	Description	Provider
`intrstat`	Report interrupt statistics (Solaris binary DTrace consumer)	sdt
`lockstat`	Report kernel lock and profile statistics (binary DTrace consumer)	lockstat
`koncpu.d`	Profile kernel on-CPU stacks	profile
`koffcpu.d`	Count kernel off-CPU stacks by time	sched
`taskq.d`	Measure task queue wait and execution time (Solaris)	sdt
`priclass.d`	Priority distribution by scheduling class	profile
`cswstat.d`	Context switch time statistics	sched
`putnexts.d`	stream `putnext()` tracing with stack backtraces	fbt

still observe this activity on Mac OS X via DTrace using the fbt or profile providers; you just don't have the neat summaries that `intrstat(1M)` provides.)

Script

`intrstat(1M)` is a binary executable that uses DTrace directly via libdtrace (instead of a script that uses libdtrace indirectly via `dtrace(1M)` and the D language).

Examples

On a four-CPU system, `intrstat(1M)` was used to examine network interrupts during load:

```
solaris# intrstat 1

      device |      cpu0 %tim      cpu1 %tim      cpu2 %tim      cpu3 %tim
-------------+------------------------------------------------------------
    e1000g#0 |         0  0.0         0  0.0       130  2.6         0  0.0

      device |      cpu0 %tim      cpu1 %tim      cpu2 %tim      cpu3 %tim
-------------+------------------------------------------------------------
    e1000g#0 |         0  0.0         0  0.0       220  5.0         0  0.0

      device |      cpu0 %tim      cpu1 %tim      cpu2 %tim      cpu3 %tim
-------------+------------------------------------------------------------
    e1000g#0 |         0  0.0         0  0.0      1154 32.0         0  0.0
       mpt#0 |         0  0.0         0  0.0        95  0.6         0  0.0

      device |      cpu0 %tim      cpu1 %tim      cpu2 %tim      cpu3 %tim
-------------+------------------------------------------------------------
    e1000g#0 |         0  0.0         0  0.0      1231 41.3         0  0.0
       mpt#0 |         0  0.0         0  0.0         0  0.0         0  0.0
      ohci#0 |         0  0.0        17  0.0         0  0.0         0  0.0
      ohci#1 |         0  0.0        17  0.1         0  0.0         0  0.0
```

Output is printed every second. The last output summary shows the e1000g device (1 Gbit/sec Ethernet interface) interrupts were consuming more than 40 percent of CPU 2. e1000g is mapped to CPU 2 on this system:

```
solaris# mdb -k
Loading modules: [ unix krtld genunix specfs dtrace cpu.AuthenticAMD.15 uppc ... ]
> ::interrupts
IRQ  Vector IPL Bus   Type  CPU Share APIC/INT# ISR(s)
1    0x41   5   ISA   Fixed 0   1     0x0/0x1   i8042_intr
4    0xb0   12  ISA   Fixed 3   1     0x0/0x4   asyintr
9    0x81   9   PCI   Fixed 1   1     0x0/0x9   acpi_wrapper_isr
12   0x42   5   ISA   Fixed 0   1     0x0/0xc   i8042_intr
19   0x20   1   PCI   Fixed 1   2     0x0/0x13  ohci_intr, ohci_intr
24   0x62   6   PCI   Fixed 2   1     0x1/0x0   e1000g_intr
25   0x63   6   PCI   Fixed 2   1     0x1/0x1   e1000g_intr
26   0x60   6   PCI   Fixed 2   1     0x1/0x2   e1000g_intr
27   0x61   6   PCI   Fixed 2   1     0x1/0x3   e1000g_intr
28   0x40   5   PCI   Fixed 2   1     0x2/0x0   mpt_intr
[...]
```

The reason it consumes more than 40 percent can be determined in a number of ways using DTrace, such as profiling the kernel stack, as shown in the one-liners.

lockstat

`lockstat(1M)` is a powerful tool on Solaris and FreeBSD to examine kernel lock events, such as spin, block, and hold time. It can also profile (sample) kernel activity. `lockstat(1M)` existed on Solaris before DTrace[5] and was used as a static kernel framework to retrieve this data; with Solaris 10, `lockstat(1M)` became DTrace-based.

Script

`lockstat(1M)` is a binary executable that dynamically produces a D script that is sent to libdtrace (instead of a static D script sent to libdtrace via `dtrace(1M)`). If it is of interest, this D script can be examined using the `-V` option:

```
solaris# lockstat -V sleep 5
lockstat: vvvv D program vvvv
lockstat:::adaptive-spin
{
        @avg[0ULL, (uintptr_t)arg0, caller] = avg(arg1);
}

lockstat:::adaptive-block
{
        @avg[1ULL, (uintptr_t)arg0, caller] = avg(arg1);
}
[...output truncated...]
```

Examples

Examples include usage, default output, stacks, and profiling with stack.

Usage

Use the `-h` option to see usage:

```
solaris# lockstat -h
Usage: lockstat [options] command [args]

Event selection options:
```

5. This was created by Jeff Bonwick, coinventor of ZFS, inventor of kernel slab allocation, and so on.

```
  -C              watch contention events [on by default]
  -E              watch error events [off by default]
  -H              watch hold events [off by default]
  -I              watch interrupt events [off by default]
  -A              watch all lock events [equivalent to -CH]
  -e event_list   only watch the specified events (shown below);
                  <event_list> is a comma-separated list of
                  events or ranges of events, e.g. 1,4-7,35
  -i rate         interrupt rate for -I [default: 97 Hz]

Data gathering options:

  -b              basic statistics (lock, caller, event count)
  -t              timing for all events [default]
  -h              histograms for event times
  -s depth        stack traces <depth> deep
  -x opt[=val]    enable or modify DTrace options

Data filtering options:

  -n nrecords     maximum number of data records [default: 10000]
  -l lock[,size]  only watch <lock>, which can be specified as a
                  symbolic name or hex address; <size> defaults
                  to the ELF symbol size if available, 1 if not
  -f func[,size]  only watch events generated by <func>
  -d duration     only watch events longer than <duration>
  -T              trace (rather than sample) events
[...]
```

Commonly used options include `-C` (which is on by default anyway) to watch lock contention events, `-s` to include stacks, and `-n` to increase the records (in response to `lockstat(1M)` warnings about dropping events).

Default Output

By default, `lockstat(1M)` will examine lock contention events. A command is passed for `lockstat(1M)` that it will execute and wait for completion, so you can specify an interval by passing the `sleep(1)` command. Regardless of the command, events are collected on a systemwide basis.

Here the `-o` option is also used to write to an output file, because the output is often many pages long (beware: `-o` appends, not overwrites):

```
solaris# lockstat -o lockstat.out sleep 5
solaris# more lockstat.out

Adaptive mutex spin: 20028 events in 5.041 seconds (3973 events/sec)

Count indv cuml rcnt     nsec Lock                   Caller
-------------------------------------------------------------------------------
 1276   6%   6% 0.00    15985 0xffffff821f761c48     taskq_thread+0x26c
 1247   6%  13% 0.00     1621 0xffffff8221fa6c50     dsl_dataset_block_kill+0x1af
 1190   6%  19% 0.00     2790 0xffffff821f761a18     taskq_thread+0x26c
 1096   5%  24% 0.00     3912 0xffffff81eb648020     mpt_scsi_start+0x9d
 1062   5%  29% 0.00     4502 0xffffff821f761c48     taskq_dispatch+0x2ea
  393   2%  31% 0.00    35982 0xffffff821f761c48     cv_wait+0x69
```

continues

```
  368   2%   33% 0.00     1100 0xffffff8221fa6c50     dsl_dir_tempreserve_clear+0x70
  365   2%   35% 0.00     1276 0xffffff8221fa6c50     dsl_dir_willuse_space_impl+0x35
  353   2%   37% 0.00     1958 0xffffff81eb648020     mpt_intr+0x5d
[...output truncated...]
```

`nsec` is the average duration of the events in nanoseconds, and `Count` shows the number of events. The top lock was from `taskq_thread()`, with 1,276 spin events at 15,985 ns per event, making a total of 20.3 ms of CPU time spent spinning on this lock.

Stacks

The stack backtrace for lock events reveals the code path responsible and can be included in the output with the `-s` option. Here, five levels of stack trace are included (`-s5`):

```
solaris# lockstat -s5 -o lockstat.out sleep 5
solaris# more lockstat.out

Adaptive mutex spin: 38423 events in 5.041 seconds (7622 events/sec)

-------------------------------------------------------------------------------
Count indv cuml rcnt     nsec Lock                      Caller
 4309  11%  11% 0.00     7244 vph_mutex+0x8000          page_hashin+0xb4

      nsec ------ Time Distribution ------ count     Stack
       512 |                               69        page_create_io+0x2c7
      1024 |@@@@@@@@                       1252      page_create_io_wrapper+0x57
      2048 |@@@@@@@@@                      1295      segkmem_xalloc+0xc0
      4096 |@@@@@                          822       segkmem_alloc_io_2G+0x3b
      8192 |@@@                            466
     16384 |@                              179
     32768 |                               78
     65536 |                               58
    131072 |                               30
    262144 |                               44
    524288 |                               14
   1048576 |                               2
[...]
```

The stack trace for this lock event showed that it originated from `segkmem`. The distribution plot shown by `lockstat(1M)` should be familiar by now, because this was the inspiration for the way DTrace prints quantize aggregations.

Profiling with Stack

`lockstat(1M)` can also profile the kernel, sampling at a specified rate. Before DTrace, this was the best way to determine kernel CPU time. Here `-s5` is used to also record five levels of stack trace:

```
solaris# lockstat -Ii997 -s5 -o profile.out sleep 5
solaris# more profile.out
Count indv cuml rcnt     nsec CPU+PIL                Caller
  105   0%  87% 0.00    62312 cpu[0]                 mmu_tlbflush_entry+0x3

      nsec ------ Time Distribution ------ count     Stack
     32768 |@@@@@@@@@@@@@@@@@@@@           71        hat_tlb_inval+0x312
     65536 |@@@@@@@@                       30        x86pte_inval+0xaa
    131072 |                               0         hat_pte_unmap+0xfd
    262144 |                               0         hat_unload_callback+0x193
    524288 |                               0
   1048576 |@                              4

[...]
```

The hottest on-CPU stack trace was from HAT performing a translation look-aside buffer (TLB) flush. The output contains multiple groups of results such as that listed previously and is sorted from most to least frequent.

koncpu.d

This is a script version of the DTrace one-liner to profile the kernel (see the "One-Liners" section for examples). As one of the most useful and frequently used one-liners, it may save typing to provide it as a script, where it can also be more easily enhanced.

Script

```
#!/usr/sbin/dtrace -s

profile:::profile-1001
{
        @["\n  on-cpu stack (count @1001hz):", stack()] = count();
}
```

Script koncpu.d

Example

The output now includes the frequency rate:

```
solaris# koncpu.d
dtrace: script 'koncpu.d' matched 1 probe
^C
[...output truncated...]

  on-cpu stack (count @1001hz):
              unix`mach_cpu_idle+0x6
              unix`cpu_idle+0xaf
              unix`cpu_idle_adaptive+0x19
              unix`idle+0x114
              unix`thread_start+0x8
            10215
```

As a script, additional DTrace actions can be added at the command line. Here it is directed to exit after ten seconds and save the output to a file:

```
solaris# koncpu.d -n 'tick-10sec { exit(0); }' -o profile.out
dtrace: script 'koncpu.d' matched 1 probe
dtrace: description 'tick-10sec ' matched 1 probe
```

This output file now contains both the interval time and the frequency rate, which are useful reminders to take into consideration when later interpreting the output.

```
solaris# cat profile.out
CPU     ID                    FUNCTION:NAME
  8  17724                      :tick-10sec


  on-cpu stack (count @1001hz):
              unix`page_nextn+0x4a
              genunix`fsflush_do_pages+0x104
[...]
```

koffcpu.d

As a companion to `koncpu.d`, the `koffcpu.d` script measures the time spent off-CPU by stack trace. This time includes device I/O, lock wait, and dispatcher queue latency, and as such `koffcpu.d` could be a useful script (see the following example).

Script

The script saves a thread-local time stamp when a thread leaves a CPU and then calculates the delta time when it returns. The kernel stack trace is included in the output.

```
#!/usr/sbin/dtrace -s

sched:::off-cpu
{
        self->start = timestamp;
}

sched:::on-cpu
/self->start/
{
        this->delta = (timestamp - self->start) / 1000;
        @["off-cpu (us):", stack()] = quantize(this->delta);
        self->start = 0;
}
```
Script koffcpu.d

Example

This was executed on Solaris. The output shows that the longest off-CPU stack traces correspond to the generic `taskq_thread()`. Many code paths that leave the CPU and wait for I/O are processed as tasks by such asynchronous task threads, making the stack trace rather uninteresting. To identify the stacks affected by this latency, more DTrace is required, such as tracing when work is added to a task queue and when it completes (see `taskq.d`).

```
solaris# koffcpu.d
dtrace: script 'koffcpu.d' matched 6 probes
^C
[...output truncated...]

  off-cpu (us):
              genunix`cv_wait+0x61
              genunix`taskq_thread_wait+0x84
              genunix`taskq_thread+0x2d1
              unix`thread_start+0x8

           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@                                        1
            8192 |                                         0
           16384 |@@@@@@@@@@@@@                            22
           32768 |@@@@@@@@@@@@@                            22
           65536 |@@@@@                                    8
          131072 |@@@@@@                                   11
          262144 |@@                                       3
          524288 |@                                        1
         1048576 |                                         0
```

`koffcpu.d` may still identify latency by stack trace in some cases; however, it may be more interesting as an example of DTrace exposing the complexities of reality.

taskq.d

As shown in the example of `koffcpu.d`, functions can be handled by task queues in the kernel. Task queues are used for a number of reasons; this quotation is from the Solaris man page `taskq(9F)`:

> A kernel task queue is a mechanism for general-purpose asynchronous task scheduling that enables tasks to be performed at a later time by another thread. There are several reasons why you may utilize asynchronous task scheduling:
>
> 1. You have a task that isn't time-critical, but a current code path that is.
> 2. You have a task that may require grabbing locks that a thread already holds.
> 3. You have a task that needs to block (for example, to wait for memory) but have a thread that cannot block in its current context.

4. You have a code path that can't complete because of a specific condition but also can't sleep or fail. In this case, the task is immediately queued and then is executed after the condition disappears.
5. A task queue is just a simple way to launch multiple tasks in parallel.

The `taskq.d` script provides statistics on task queue operations: task dispatch count, total task wait time, and total task execution time.

Script

Task queues have a standard interface as part of DDI (device drivers interface), making an enticing source for fbt probes. However, on Solaris, sdt probes are placed for taskq analysis:[6]

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN { trace("Tracing...  Interval 10 seconds, or Ctrl-C.\n"); }

sdt:::taskq-enqueue
{
        this->tq  = (taskq_t *)arg0;
        this->tqe = (taskq_ent_t *) arg1;
        @c[this->tq->tq_name, this->tqe->tqent_func] = count();
        time[arg1] = timestamp;
}

sdt:::taskq-exec-start
/time[arg1]/
{
        this->wait = timestamp - time[arg1];
        this->tq  = (taskq_t *)arg0;
        this->tqe = (taskq_ent_t *) arg1;
        @w[this->tq->tq_name, this->tqe->tqent_func] = sum(this->wait);
        time[arg1] = timestamp;
}

sdt:::taskq-exec-end
/time[arg1]/
{
        this->exec = timestamp - time[arg1];
        this->tq  = (taskq_t *)arg0;
        this->tqe = (taskq_ent_t *) arg1;
        @e[this->tq->tq_name, this->tqe->tqent_func] = sum(this->exec);
        time[arg1] = 0;
}

profile:::tick-10s,
dtrace:::END
{
        normalize(@w, 1000000);
        normalize(@e, 1000000);
```

6. *http://blogs.sun.com/akolb/entry/task_queues_in_opensolaris*

```
        printf("\n %-22s %-25s %8s %9s %9s\n", "TASKQ NAME", "FUNCTION",
            "COUNT", "T_WAITms", "T_EXECms");
        printa(" %-22.22s %-25.25a %8@d %@9d %@9d\n", @c, @w, @e);
        trunc(@c); trunc(@w); trunc(@e);
}
```

Script tsskq.d

These sdt probes are not currently available on Mac OS X or FreeBSD; fbt tracing can be used instead.

Example

While tracing, a ZFS write workload was performed. The total time spent waiting and executing on the task queues can be seen. To see why these functions are being placed on a task queue, the kernel stack trace can be examined during `sdt:::taskq-enqueue`.

```
solaris# taskq.d
Tracing...  Interval 10 seconds, or Ctrl-C.

 TASKQ NAME             FUNCTION                     COUNT  T_WAITms  T_EXECms
 kmem_taskq             genunix`kmem_update_timeo        1         0         0
 kmem_taskq             genunix`kmem_cache_scan          1         0         0
 zil_clean              zfs`zil_itx_clean                6         0        18
 kmem_taskq             genunix`kmem_hash_rescale       17         0         0
 timeout_taskq          genunix`timeout_execute         40         0         0
 zio_null_intr          zfs`zio_execute                144         1         1
 zio_null_issue         zfs`zio_execute                144        36         3
 zio_ioctl_intr         zfs`zio_execute                156         0         1
 cpudrv_cpudrv_monitor  cpudrv`cpudrv_monitor          160         1         2
 callout_taskq          genunix`callout_execute       1416        90       176
 zio_write_issue        zfs`zio_execute              17390     95569      5450
 zio_write_intr         zfs`zio_execute              34855      4782      2196
```

priclass.d

DTrace provides excellent visibility into the kernel thread scheduler, not just from the sched provider but also from the profile provider, by sampling what threads are on-CPU and with what priority. The `priclass.d` script samples the scheduling class and priority value.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
```

continues

```
{
        printf("Sampling... Hit Ctrl-C to end.\n");
}

profile:::profile-1001hz
{
        @count[stringof(curlwpsinfo->pr_clname)]
            = lquantize(curlwpsinfo->pr_pri, 0, 170, 10);
}
```
Script priclass.d

Example

Various applications were executed on Solaris in different scheduling classes. These included a windowing environment (runs in IA, interactive), `prstat -R` (runs in RT, real time), and a CPU busy process (runs in TS, time sharing). The kernel will also be running (runs in SYS, system):

```
solaris# priclass.d
Sampling... Hit Ctrl-C to end.
^C

  IA
           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 30
              60 |                                         0

  SYS
           value  ------------- Distribution ------------- count
             < 0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  4959
               0 |                                         0
              10 |                                         0
              20 |                                         0
              30 |                                         0
              40 |                                         0
              50 |                                         0
              60 |                                         30
              70 |                                         0
              80 |                                         0
              90 |                                         0
             100 |                                         0
             110 |                                         0
             120 |                                         0
             130 |                                         0
             140 |                                         0
             150 |                                         0
             160 |                                         50
          >= 170 |                                         0

  RT
           value  ------------- Distribution ------------- count
              90 |                                         0
             100 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 110
             110 |                                         0

  TS
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@                          2880
              10 |@@@@@@@                                  1280
```

```
          20 |@@@@@                                    990
          30 |@@@@@                                    920
          40 |@@@@                                     670
          50 |@@@@                                     730
          60 |                                         0
```

The priority numbers match those expected for the classes (see Chapter 3 in *Solaris Internals* [McDougall and Mauro, 2006]). The distribution plot for the `TS` class also matches expected changes in priority for CPU busy processes (also see `ts_dptbl(4)`), which demotes the priority as the thread uses its quantum, until the thread is promoted again back to top priority (because of `ts_maxwait`).

cswstat.d

Thread context switch time was once a concern for performance analysis, because excessive switching could consume a significant amount of CPU. This was previously analyzed by using microbenchmarks to determine the expected context switch time and then measuring the number of current context switches with tools such as `vmstat(1M)` and `mpstat(1M)`. These days, the context switch time is much faster with respect to the clock speed of CPUs, and it has become less of a concern. However, it does remain an interesting topic for analysis, and DTrace is able to measure context switch time directly on the real target workload by using the sched provider.

Script

The script measures the time between descheduling one thread and scheduling the next:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

dtrace:::BEGIN
{
        /* print header */
        printf("%-20s  %8s %12s %12s\n", "TIME", "NUM", "CSWTIME(us)",
            "AVGTIME(us)");
        times = 0;
        num = 0;
}

sched:::off-cpu
{
        /* csw start */
        num++;
        start[cpu] = timestamp;
}

```

```
21  sched:::on-cpu
22  /start[cpu]/
23  {
24          /* csw end */
25          times += timestamp - start[cpu];
26          start[cpu] = 0;
27  }
28
29  profile:::tick-1sec
30  {
31          /* print output */
32          printf("%20Y  %8d %12d %12d\n", walltimestamp, num, times/1000,
33              num == 0 ? 0 : times/(1000 * num));
34          times = 0;
35          num = 0;
36  }
```

Script cswstat.d

Line 17 increments a scalar global variable `num`; these are usually better served as aggregations (for example, `@num = count()`). However, `num` is later read on line 33 as part of a formula, which an aggregation cannot do (they can only be printed). This also applies to the `times` variable, which is also a scalar global.

Example

On this system, the average context switch time is two microseconds, and the total time spent is about three milliseconds (total across all 24 CPUs in this system) per second, which is negligible.

```
solaris# cswstat.d
TIME                         NUM  CSWTIME(us)  AVGTIME(us)
2010 Jun  3 02:05:24        1291         3139            2
2010 Jun  3 02:05:25        1069         2197            2
2010 Jun  3 02:05:26        1863         4040            2
2010 Jun  3 02:05:27        1102         2659            2
2010 Jun  3 02:05:28        1889         4152            2
2010 Jun  3 02:05:29         871         1805            2
[...]
```

putnexts.d

The `STREAMS` interface is used on Solaris to deliver messages around the kernel and across kernel modules via queues. Although it's recently been removed from some hot code-paths (in TCP/IP), there are still many modules that operate via streams. This operation can be viewed by tracing the STREAMS API functions, such as `putnext()`, which puts a message to a kernel module queue.

Script

Although this is an fbt-based script and considered unstable, it's also very short and should ideally be easy to maintain to match changes in the kernel:

```
#!/usr/sbin/dtrace -s

fbt::putnext:entry
{
        @[stringof(args[0]->q_qinfo->qi_minfo->mi_idname), stack(5)] = count();
}
```

Script putnexts.d

The kernel stack trace is capped to the first five stack frames.

Example

```
solaris# putnexts.d
dtrace: script 'putnexts.d' matched 1 probe
^C
[...output truncated...]
  arp
              ip`arp_output+0x2f1
              ip`arp_request+0xf6
              ip`nce_timer+0x619
              genunix`callout_list_expire+0x77
              genunix`callout_expire+0x31
               17
  nxge
              dld`dld_str_rx_raw+0xbf
              dls`dls_rx_promisc+0x181
              mac`mac_promisc_dispatch_one+0x94
              mac`mac_promisc_dispatch+0x110
              mac`mac_rx_common+0x3e
               73
  strwhead
              genunix`strput+0x19d
              genunix`strputmsg+0x2a0
              genunix`msgio32+0x202
              genunix`putmsg32+0x78
              unix`sys_syscall32+0xff
               83
  tcp
              ip`tcp_input_data+0x3398
              ip`squeue_enter+0x440
              ip`ip_fanout_v4+0x48d
              ip`ire_recv_local_v4+0x366
              ip`ill_input_short_v4+0x69e
              937
```

The first stack shows the ip module passing a message to arp for processing. The most frequent was shown at the bottom, ip passing messages to tcp.

Summary

DTrace provides an incredibly detailed view of kernel internal operation, which can be a crucial capability, whether the goal is to root-cause a performance problem, to understand capacity, or simply to study operating system internals. There are a broad range of DTrace providers and methods for examining the kernel, enabling time- and count-based profiles, measuring kernel function times, examining arguments and return values, and tracing kernel code flow. These were demonstrated throughout this chapter.

13

Tools

This chapter describes some tools built using DTrace to collect data; several such have emerged since DTrace was first made available with the initial release of Solaris 10. This chapter is not intended to provide information on how to use the various tools discussed here; each is very well documented with numerous examples, all easily available online. Our goal here is to briefly introduce these tools to the reader. All the tools discussed in this chapter are easily and freely downloadable if you want to explore them in more detail.

The tools discussed in this chapter are as follows.

The DTraceTookit: This is a huge collection of tools implemented as scripts.

Chime: This is a standalone tool for visualizing DTrace aggregations. It is also a component of the DTrace GUI plug-in for NetBeans and Sun Studio.

DTrace GUI plug-in for NetBeans and Sun Studio: This integrates the DTraceToolkit and Chime into NetBeans and Sun Studio.

DLight: This is part of the Sun Studio GUI tools for doing application analysis based on DTrace.

Mac OS X Instruments: This is part of the Mac OS X developer tools, a GUI-based tool that uses DTrace for application analysis.

Analytics: This is a powerful Web-based graphical tool that ships with the Oracle ZFS Storage product family.

The DTraceToolkit

The DTraceToolkit is a collection of more than 200 DTrace scripts and one-liners for performance observability and troubleshooting. It is designed to serve both as a toolkit of prewritten scripts and as a set of examples from which to learn DTrace. The following are the major components of the toolkit:

Scripts

A man page for every script

An examples file for every script

The man(ual) pages document the purpose of the scripts, their output, and any command-line options; the example files demonstrate using the scripts and show how to read and interpret their output. Since the scripts are designed to be intuitive, the examples file—which contains the CLI equivalent of screenshots—is an effective form of documentation, because the output for some scripts alone may be self-evident.

Most of the development time for the DTraceToolkit is spent testing scripts for different workloads on different operating systems.[1] It can be easy to write scripts that appear to work, but it's much harder to develop scripts that are proven to work. Although testing is essential, some scripts cannot be tested fully. fbt provider–based scripts should be tested on every possible version of the kernel, but there are so many that this has become impractical. The use of the fbt provider in the DTraceToolkit is therefore deliberately minimized because of its instability. Chapter 12, Kernel, discusses the stability issues with using the fbt provider in detail.

The DTraceToolkit is an open source project and can be downloaded free of charge. It is not supported by any company.

Locations

The DTraceToolkit is currently available from a few locations:

http://sourceforge.net/projects/dtracetoolkit

www.brendangregg.com/dtrace.html#DTraceToolkit

http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit

It was also shipped in OpenSolaris in `/opt/DTT`, and 44 scripts are in Mac OS X in `/usr/bin`.

1. Stefan Parvu performs testing for each release using an automated test harness.

Versions

The DTraceToolkit was created in May 2005 by Brendan Gregg, who also wrote most of the scripts throughout this book. Several versions have been released, the latest being version 0.99 released in September 2007, containing 230 scripts. Another version is planned after the release of this book.

Although it was initially aimed at the Solaris and OpenSolaris systems, with the inclusion of DTrace in Mac OS X and FreeBSD, the toolkit is moving toward more directly supporting other operating systems. Many of the scripts, in particular those based on stable DTrace providers, work across different operating systems without changes; some of the scripts require only minor changes to become generic and avoid Solaris-isms (for example, when using shell wrappers: picking `/bin/sh` instead of `/usr/bin/sh`). When Apple included 44 scripts in Mac OS X, some were modified so that they executed properly.

Installation

The DTraceToolkit is shipped as a compressed tar file. It can be expanded using the following:

```
# gzcat DTraceToolkit-0.99.tar.gz | tar xvf -
```

At this point, the scripts can be executed and the documentation read. If desired, an installer script (called `install`) can be executed, which copies the toolkit to `/opt/DTT`.

Scripts

Table 13-1 summarizes the scripts in the DTraceToolkit, the subdirectory under which they are provided, and a description of their purpose. A summary version of this table is listed in the DTraceToolkit under `Docs/Contents`.

Table 13-1 DTraceToolkit 0.99 Scripts

Script	Directory	Description
`dexplorer`	/	Run a series of scripts and archive output.
`dtruss`	/	Process syscall info. DTrace truss.
`dvmstat`	/	vmstat by PID/name/command.

continues

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`errinfo`	`/`	Report syscall failures with details.
`execsnoop`	`/`	Snoop process execution as it occurs.
`iosnoop`	`/`	Snoop I/O events as they occur.
`iopattern`	`/`	Print disk I/O pattern.
`iotop`	`/`	Display top disk I/O events by process.
`opensnoop`	`/`	Snoop file opens as they occur.
`procsystime`	`/`	Analyze process system call times.
`rwsnoop`	`/`	Snoop read/write events.
`rwtop`	`/`	Display top read/write bytes by process.
`statsnoop`	`/`	Snoop file stats as they occur.
`httpdstat.d`	`Apps`	Realtime httpd statistics.
`nfswizard.d`	`Apps`	NFS client activity wizard.
`shellsnoop`	`Apps`	Snoop live shell activity.
`weblatency.d`	`Apps`	Web site latency statistics.
`cputypes.d`	`Cpu`	List CPU types.
`cpuwalk.d`	`Cpu`	Measure which CPUs a process runs on.
`dispqlen.d`	`Cpu`	Dispatcher queue length by CPU.
`intbycpu.d`	`Cpu`	Interrupts by CPU.
`intoncpu.d`	`Cpu`	Interrupt on-cpu usage.
`inttimes.d`	`Cpu`	Interrupt on-cpu time total.
`loads.d`	`Cpu`	Print load averages.
`runocc.d`	`Cpu`	Run queue occupancy by CPU.
`xcallsbypid.d`	`Cpu`	CPU cross calls by PID.
`bitesize.d`	`Disk`	Print disk event size report.
`diskhits`	`Disk`	Disk access by file offset.
`hotspot.d`	`Disk`	Print disk event by location.
`iofile.d`	`Disk`	I/O wait time by filename and process.
`iofileb.d`	`Disk`	I/O bytes by filename and process.
`iopending`	`Disk`	Plot number of pending disk events.
`pathopens.d`	`Disk`	Pathnames successfully opened count.
`seeksize.d`	`Disk`	Print disk seek size report.
`fsrw.d`	`FS`	File system read/write event tracing.
`fspaging.d`	`FS`	File system read/write and paging tracing.

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`rfsio.d`	`FS`	Read FS I/O stats, with cache miss rate.
`rfileio.d`	`FS`	Read file I/O stats, with cache miss rate.
`vopstat`	`FS`	vnode interface statistics.
`j_calldist.d`	`Java`	Measure Java elapsed times for different types of operations.
`j_calls.d`	`Java`	Count Java calls (method, and so on) using DTrace.
`j_calltime.d`	`Java`	Measure Java elapsed times for different types of operations.
`j_classflow.d`	`Java`	Trace a Java class method flow using DTrace.
`j_cpudist.d`	`Java`	Measure Java on-CPU times for different types of operations.
`j_cputime.d`	`Java`	Measure Java on-CPU times for different types of operations.
`j_events.d`	`Java`	Count Java events using DTrace.
`j_flow.d`	`Java`	Snoop Java execution showing method flow using DTrace.
`j_flowtime.d`	`Java`	Snoop Java execution with method flow and delta times.
`j_methodcalls.d`	`Java`	Count Java method calls DTrace.
`j_objnew.d`	`Java`	Report Java object allocation using DTracev
`j_package.d`	`Java`	Count Java class loads by package using DTracev
`j_profile.d`	`Java`	Sample stack traces with Java translations using DTrace.
`j_stat.d`	`Java`	Java operation stats using DTrace.
`j_syscalls.d`	`Java`	Count Java methods and syscalls using DTrace.
`j_syscolors.d`	`Java`	Trace Java method flow plus syscalls, in color.
`j_thread.d`	`Java`	Snoop Java thread execution using DTrace.
`j_who.d`	`Java`	Trace Java calls by process using DTrace.
`js_calldist.d`	`JavaScript`	Measure JavaScript elapsed times for types of operations.
`js_calls.d`	`JavaScript`	Count JavaScript calls using DTrace.
`js_calltime.d`	`JavaScript`	Measure JavaScript elapsed times for types of operations.

continues

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
js_cpudist.d	JavaScript	Measure JavaScript on-CPU times for types of operations.
js_cputime.d	JavaScript	Measure JavaScript on-CPU times for types of operations.
js_execs.d	JavaScript	JavaScript execute snoop using DTrace.
js_flow.d	JavaScript	Snoop JavaScript execution showing function flow using DTrace.
js_flowinfo.d	JavaScript	JavaScript function flow with info using DTrace.
js_flowtime.d	JavaScript	JavaScript function flow with delta times using DTrace.
js_objcpu.d	JavaScript	Measure JavaScript object creation on-CPU time using DTrace.
js_objgc.d	JavaScript	Trace JavaScript Object GC using DTrace.
js_objnew.d	JavaScript	Count JavaScript object creation using DTrace.
js_stat.d	JavaScript	JavaScript operation stats using DTrace.
js_who.d	JavaScript	Trace JavaScript function execution by process using DTrace.
cputimes	Kernel	Print time by kernel/idle/process.
cpudists	Kernel	Time distribution by kernel/idle/process.
cswstat.d	Kernel	Context switch time statistics.
dnlcps.d	Kernel	DNLC stats by process.
dnlcsnoop.d	Kernel	Snoop DNLC activity.
dnlcstat	Kernel	DNLC statistics.
kstat_types.d	Kernel	Trace `kstat` reads with type info.
modcalls.d	Kernel	Kernel function calls by module name.
priclass.d	Kernel	Priority distribution by scheduling class.
pridist.d	Kernel	Process priority distribution.
putnexts.d	Kernel	Trace who is putting to which streams module.
whatexec.d	Kernel	Examine the type of files executed.
lockbyproc.d	Locks	Lock time by process name.
lockbydist.d	Locks	Lock time distribution by process name.
anonpgpid.d	Mem	Anonymous memory paging info by PID on-CPU.
minfbypid.d	Mem	Minor faults by PID.
minfbyproc.d	Mem	Minor faults by process name.

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`pgpginbypid.d`	`Mem`	Pages paged in by PID.
`pgpginbyproc.d`	`Mem`	Pages paged in by process name.
`swapinfo.d`	`Mem`	Print virtual memory info.
`vmbypid.d`	`Mem`	Virtual memory stats by PID.
`vmstat.d`	`Mem`	`vmstat` demo using DTrace.
`vmstat-p.d`	`Mem`	`vmstat -p` demo using DTrace.
`xvmstat`	`Mem`	Extended vmstat demo using DTrace.
`guess.d`	`Misc`	Guessing game.
`wpm.d`	`Misc`	Words per minute tracing.
`woof.d`	`Misc`	Audio alert for new processes.
`connections`	`Net`	Print inbound TCP connections by process.
`icmpstat.d`	`Net`	Print ICMP statistics.
`tcpsnoop`	`Net`	Snoop TCP network packets by process, Solaris 10 3/05.
`tcpsnoop_snv`	`Net`	Snoop TCP network packets by process, Solaris Nevada.
`tcpsnoop.d`	`Net`	Snoop TCP network packets by process, Solaris 10 3/05.
`tcpsnoop_snv.d`	`Net`	Snoop TCP network packets by process, Solaris Nevada.
`tcpstat.d`	`Net`	Print TCP statistics.
`tcptop`	`Net`	Display top TCP network packets by PID, Solaris 10 3/05.
`tcoptop_snv`	`Net`	Display top TCP network packets by PID, Solaris Nevada.
`tcpwdist.d`	`Net`	Simple TCP write distribution by process.
`udpstat.d`	`Net`	Print UDP statistics.
`pl_calldist.d`	`Perl`	Measure Perl elapsed times for subroutines.
`pl_calltime.d`	`Perl`	Measure Perl elapsed times for subroutines.
`pl_cpudist.d`	`Perl`	Measure Perl on-CPU times for subroutines.
`pl_cputime.d`	`Perl`	Measure Perl on-CPU times for subroutines.
`pl_flow.d`	`Perl`	Snoop Perl execution showing subroutine flow.
`pl_flowinfo.d`	`Perl`	Snoop Perl subroutine flow with info using DTrace.

continues

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`pl_flowtime.d`	`Perl`	Snoop Perl subroutines with flow and delta times.
`pl_malloc.d`	`Perl`	Perl libc malloc analysis.
`pl_subcalls.d`	`Perl`	Measure Perl subroutine calls using DTrace.
`pl_syscalls.d`	`Perl`	Count Perl subroutine calls and syscalls using DTrace.
`pl_syscolors.d`	`Perl`	Trace Perl subroutine flow plus syscalls, in color.
`pl_who.d`	`Perl`	Trace Perl subroutine execution by process using DTrace.
`php_calldist.d`	`Php`	Measure PHP elapsed times for functions.
`php_calltime.d`	`Php`	Measure PHP elapsed times for functions.
`php_cpudist.d`	`Php`	Measure PHP on-CPU times for functions.
`php_cputime.d`	`Php`	Measure PHP on-CPU times for functions.
`php_flow.d`	`Php`	Snoop PHP execution showing function flow.
`php_flowinfo.d`	`Php`	Snoop PHP function flow with info using DTrace.
`php_flowtime.d`	`Php`	Snoop PHP functions with flow and delta times.
`php_funccalls.d`	`Php`	Measure PHP function calls using DTrace.
`php_malloc.d`	`Php`	PHP libc malloc analysis.
`php_syscalls.d`	`Php`	Count PHP function calls and syscalls using DTrace.
`php_syscolors.d`	`Php`	Trace PHP function flow plus syscalls, in color.
`php_who.d`	`Php`	Trace PHP function execution by process using DTrace.
`crash.d`	`Proc`	Crashed application report.
`creatbyproc.d`	`Proc`	Snoop file `creat()` by process name.
`dappprof`	`Proc`	Profile user and lib function usage.
`dapptrace`	`Proc`	Trace user and lib function usage.
`fddist`	`Proc`	File descriptor usage distribution.
`fileproc.d`	`Proc`	Snoop files opened by process.
`kill.d`	`Proc`	Snoop process signals.
`lastwords`	`Proc`	Print syscalls before exit.
`mmapfiles.d`	`Proc`	`mmap`'d files by process.
`newproc.d`	`Proc`	Snoop new processes.
`pfilestat`	`Proc`	Show I/O latency break down by FD.
`pidpersec.d`	`Proc`	Print new PIDs per sec.

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`readbytes.d`	`Proc`	Read bytes by process name.
`readdist.d`	`Proc`	Read distribution by process name.
`rwbbypid.d`	`Proc`	Read/write bytes by PID.
`rwbypid.d`	`Proc`	Read/write calls by PID.
`rwbytype.d`	`Proc`	Read/write bytes by vnode type.
`sampleproc`	`Proc`	Sample processes on the CPUs.
`shortlived.d`	`Proc`	Check short lived process time.
`sigdist.d`	`Proc`	Signal distribution by process name.
`stacksize.d`	`Proc`	Measure stack size for running threads.
`sysbypid.d`	`Proc`	System stats by PID.
`syscallbyproc.d`	`Proc`	System calls by process name.
`syscallbypid.d`	`Proc`	System calls by process ID.
`treaded.d`	`Proc`	Sample multi-threaded CPU usage.
`topsysproc`	`Proc`	Display top syscalls by process name.
`writebytes.d`	`Proc`	Write bytes by process name.
`writedist.d`	`Proc`	Write distribution by process name.
`py_calldist.d`	`Python`	Measure Python elapsed times for functions.
`py_calltime.d`	`Python`	Measure Python elapsed times for functions.
`py_cpudist.d`	`Python`	Measure Python on-CPU times for functions.
`py_cputime.d`	`Python`	Measure Python on-CPU times for functions.
`py_flow.d`	`Python`	Snoop Python execution showing function flow.
`py_flowinfo.d`	`Python`	Snoop Python function flow with info using DTrace.
`py_flowtime.d`	`Python`	Snoop Python functions with flow and delta times.
`py_funccalls.d`	`Python`	Measure Python function calls using DTrace.
`py_malloc.d`	`Python`	Python libc malloc analysis.
`py_mallocstk.d`	`Python`	Python libc malloc analysis with full stack traces.
`py_profile.d`	`Python`	Sample stack traces with Python translations using DTrace.
`py_syscalls.d`	`Python`	Count Python function calls and syscalls using DTrace.
`py_syscolors.d`	`Python`	Trace Python function flow plus syscalls, in color.

continues

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
py_who.d	Python	Trace Python function execution by process using DTrace.
sh_calldist.d	Shell	Measure Bourne shell elapsed times for types of operations.
sh_calls.d	Shell	Count Bourne calls (func/builtin/cmd/subsh) using DTrace.
sh_calltime.d	Shell	Measure Bourne shell elapsed times for types of operations.
sh_cpudist.d	Shell	Measure Bourne shell on-CPU times for types of operations.
sh_cputime.d	Shell	Measure Bourne shell on-CPU times for types of operations.
sh_flow.d	Shell	Snoop Bourne shell execution showing function flow using .DTrace
sh_flowinfo.d	Shell	Snoop Bourne shell flow with additional info.
sh_flowtime.d	Shell	Snoop Bourne shell execution with flow and delta times.
sh_lines.d	Shell	Trace Bourne shell line execution using DTrace.
sh_pidcolors.d	Shell	Demonstration of deeper DTrace Bourne shell analysis.
sh_stat.d	Shell	Bourne shell operation stats using DTrace.
sh_syscalls.d	Shell	Count Bourne calls and syscalls using DTrace.
sh_syscolors.d	Shell	Trace Bourne shell flow plus syscalls, in color.
sh_wasted.d	Shell	Measure Bourne shell elapsed times for "wasted" commands.
sh_who.d	Shell	Trace Bourne shell line execution by process using DTrace.
sar-c.d	System	sar -c demo using DTrace.
syscallbysysc.d	System	System calls by system call.
topsyscall	System	Display top system call type.
uname-a.d	System	uname -a demo using DTrace.
tcl_calldist.d	Tcl	Measure Tcl elapsed time for different types of operations.
tcl_calls.d	Tcl	Count Tcl calls (proc/cmd) using DTrace.
tcl_calltime.d	Tcl	Measure Tcl elapsed times for different types of operations.

Table 13-1 DTraceToolkit 0.99 Scripts (*Continued*)

Script	Directory	Description
`tcl_cpudist.d`	`Tcl`	Measure Tcl on-CPU time for different types of operations.
`tcl_cputime.d`	`Tcl`	Measure Tcl on-CPU times for different types of operations.
`tcl_flow.d`	`Tcl`	Snoop Tcl execution showing procedure flow using DTrace.
`tcl_flowtime.d`	`Tcl`	Snoop Tcl execution showing procedure flow and delta times.
`tcl_ins.d`	`Tcl`	Count Tcl instructions using DTrace.
`tcl_insflow.d`	`Tcl`	Snoop Tcl execution showing procedure flow and delta times.
`tcl_methodcalls.d`	`Tcl`	Count Tcl method calls DTrace.
`tcl_procflow.d`	`Tcl`	Snoop Tcl execution showing procedure flow using DTrace.
`tcl_stat.d`	`Tcl`	Tcl operation stats using DTrace.
`tcl_syscalls.d`	`Tcl`	Count Tcl calls and syscalls using DTrace.
`tcl_syscolors.d`	`Tcl`	Trace Tcl program flow plus syscalls, in color.
`tcl_who.d`	`Tcl`	Trace Tcl calls by process using DTrace.
`setuids.d`	`User`	Snoop setuid calls.
`zvmstat`	`Zones`	`vmstat` info by zone

The best examples from this table have been included and explained in other chapters of this book. The remaining scripts are documented in the DTraceToolkit: See the script, the man page, and examples file for each.

As an example of how scripts are provided in the DTraceToolkit, the following sections list the related files for the `cpuwalk.d` script.

Script Example: cpuwalk.d

The `cpuwalk.d` script was written to help analyze the effectiveness of multithreaded applications by sampling how frequently the threads are running on different CPUs. For this to work, there needs to be a sufficiently high workload for the application to be using multiple CPUs concurrently.

An issue that this script could identify is serialization on a global lock, where only the thread holding the lock can make forward progress.

Script

The script is `Cpu/cpuwalk.d`, which is also has a symbolic link called `Bin/cpuwalk.d`, provided for convenience (all scripts are linked under `Bin` and can be searched using `grep` from one place).

```
#!/usr/sbin/dtrace -s
/*
 * cpuwalk.d - Measure which CPUs a process runs on.
 *             Written using DTrace (Solaris 10 3/05)
 *
 * This program is for multi-CPU servers, and can help identify if a process
 * is running on multiple CPUs concurrently or not.
 *
 * $Id: cpuwalk.d 3 2007-08-01 10:50:08Z brendan $
 *
 * USAGE:       cpuwalk.d [duration]
 *         eg,
 *              cpuwalk.d 10            # sample for 10 seconds
 *              cpuwalk.d               # sample until Ctrl-C is hit
 *
 * FIELDS:
 *              value           CPU id
 *              count           Number of 1000 hz samples on this CPU
 *
 * COPYRIGHT: Copyright (c) 2005 Brendan Gregg.
 *
 * CDDL HEADER START
 *
 *  The contents of this file are subject to the terms of the
 *  Common Development and Distribution License, Version 1.0 only
 *  (the "License").  You may not use this file except in compliance
 *  with the License.
 *
 *  You can obtain a copy of the license at Docs/cddl1.txt
 *  or http://www.opensolaris.org/os/licensing.
 *  See the License for the specific language governing permissions
 *  and limitations under the License.
 *
 * CDDL HEADER END
 *
 * 22-Sep-2005  Brendan Gregg   Created this.
 * 14-Feb-2006     "      "     Last update.
 */

#pragma D option quiet
#pragma D option defaultargs

inline int MAXCPUID = 1024;

dtrace:::BEGIN
{
        $1 ? printf("Sampling...\n") :
            printf("Sampling... Hit Ctrl-C to end.\n");
        seconds = 0;
}

profile:::profile-1000hz
/pid/
{
        @sample[pid, execname] = lquantize(cpu, 0, MAXCPUID, 1);
}
```

```

profile:::tick-1sec
{
        seconds++;
}

profile:::tick-1sec
/seconds == $1/
{
        exit(0);
}

dtrace:::END
{
        printa("\n     PID: %-8d CMD: %s\n%@d", @sample);
}
```

If there are concerns that the script may sample in lockstep with the application, the profile rate can be adjusted from `1000hz` to `1001hz` (or something similar).

Man Page

The manual page file is `Man/man1m/cpuwalk.d.1m`. To read it, set the `MANPATH` variable to the `Man` directory, for example: `MANPATH=/opt/DTT/Man man cpuwalk.d`.

```
Maintenance Commands                                cpuwalk.d(1m)

NAME
     cpuwalk.d - Measure which  CPUs  a  process  runs  on.  Uses
     DTrace.

SYNOPSIS
     cpuwalk.d [duration]

DESCRIPTION
     This program is for multi-CPU servers, and can help identify
     if  a  process  is  running on multiple CPUs concurrently or
     not.

     A duration may be specified in seconds.

     Since this uses DTrace, only the root user or users with the
     dtrace_kernel privilege can run this command.

OS
     Any

STABILITY
     stable.

EXAMPLES
     this runs until Ctrl-C is hit,
          # cpuwalk.d

     run for 5 seconds,
          # cpuwalk.d 5

FIELDS
     PID  process ID
```

continues

```

     CMD  process name

     value
          CPU id

     count
          number of samples (sample at 100 hz)

DOCUMENTATION
     See the DTraceToolkit for further  documentation  under  the
     Docs  directory.  The  DTraceToolkit  docs  may include full
     worked examples with  verbose  descriptions  explaining  the
     output.

EXIT
     cpuwalk.d will run until Ctrl-C  is  hit,  or  the  duration
     specified is reached.

USER COMMANDS  Last change: $Date:: 2007-08-05 #$               1

Maintenance Commands                                cpuwalk.d(1m)

SEE ALSO
     threaded.d(1M), dtrace(1M)

USER COMMANDS  Last change: $Date:: 2007-08-05 #$               2

```

Examples

The examples file is in `Examples/cpuwalk_example.txt`.

```
The following is a demonstration of the cpuwalk.d script,


cpuwalk.d is not that useful on a single CPU server,

   # cpuwalk.d
   Sampling... Hit Ctrl-C to end.
   ^C

        PID: 18843    CMD: bash

              value  ------------- Distribution ------------- count
                < 0 |                                         0
                  0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 30
                  1 |                                         0

        PID: 8079     CMD: mozilla-bin

              value  ------------- Distribution ------------- count
                < 0 |                                         0
                  0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
                  1 |                                         0

The output above shows that PID 18843, "bash", was sampled on CPU 0 a total
of 30 times (we sample at 1000 hz).



```

```
The following is a demonstration of running cpuwalk.d with a 5 second
duration. This is on a 4 CPU server running a multithreaded CPU bound
application called "cputhread",

   # cpuwalk.d 5
   Sampling...

        PID: 3        CMD: fsflush

              value  ------------- Distribution ------------- count
                  1 |                                         0
                  2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 30
                  3 |                                         0

        PID: 12186    CMD: cputhread

              value  ------------- Distribution ------------- count
                < 0 |                                         0
                  0 |@@@@@@@@@@                               4900
                  1 |@@@@@@@@@@                               4900
                  2 |@@@@@@@@@@                               4860
                  3 |@@@@@@@@@@                               4890
                  4 |                                         0

As we are sampling at 1000 hz, the application cputhread is indeed running
concurrently across all available CPUs. We measured the applicaiton on
CPU 0 a total of 4900 times, on CPU 1 a total of 4900 times, etc. As there
are around 5000 samples per CPU available in this 5 second 1000 hz sample,
the application is using almost all the CPU capacity in this server well.



The following is a similar demonstration, this time running a multithreaded
CPU bound application called "cpuserial" that has a poor use of locking
such that the threads "serialise",


   # cpuwalk.d 5
   Sampling...

        PID: 12194    CMD: cpuserial

              value  ------------- Distribution ------------- count
                < 0 |                                         0
                  0 |@@@                                      470
                  1 |@@@@@@                                   920
                  2 |@@@@@@@@@@@@@@@@@@@@@@@@@                3840
                  3 |@@@@@@                                   850
                  4 |                                         0

In the above, we can see that this CPU bound application is not making
efficient use of the CPU resources available, only reaching 3840 samples
on CPU 2 out of a potential 5000. This problem was caused by a poor use
of locks.
```

The scripts are designed to be intuitive to use and to produce output that is self-evident. This is best demonstrated in the examples files, which can often explain the function and intended usage of scripts more quickly than by reading the man page or script itself.

Chime

Chime is a graphical tool for visualizing DTrace aggregations, created by Tom Erickson and released as an OpenSolaris project. It has a library of DTrace scripts that can be navigated and executed via a graphical interface, and it allows data to be presented over time as line graphs, allowing patterns to be observed that may not be obvious at the command line. Custom DTrace scripts and one-liners can be executed, with Chime providing visualizations. Chime is also part of the DTrace GUI plug-in for NetBeans and Sun Studio.

The amount of detail shown in Figure 13-1 would require many pages of text output, which would not be as effective at showing patterns in the data.

Chime is also open source and serves as an example of the Java DTrace API, which it uses to interface to DTrace. The Java DTrace API allows Chime to read data from DTrace efficiently, rather than wrapping the text output of `dtrace(1M)`.

Locations

Chime is available on the OpenSolaris Web site: *http://hub.opensolaris.org/bin/view/Project+dtrace-chime/*.

Links to download Chime are in the install section. It is distributed as a package file that must be uncompressed (`gunzip`) and added (`pkgadd -d`).

This Web site also includes documentation for customizing and enhancing Chime by adding new "displays," which are visualizations of DTrace one-liners and

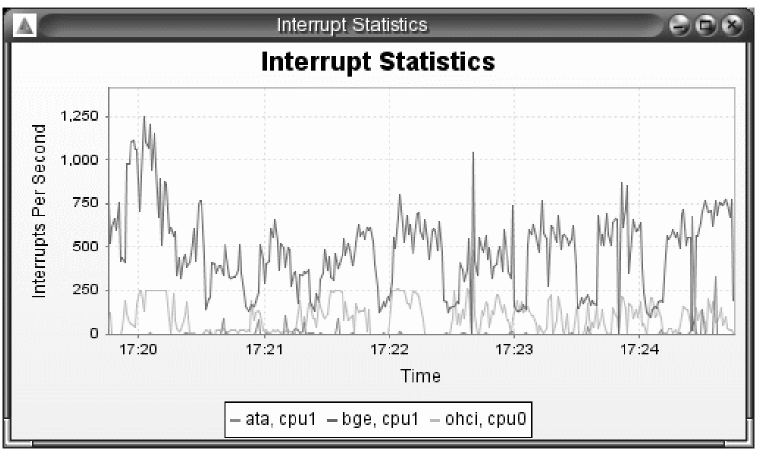


Figure 13-1 Chime displaying interrupt statistics

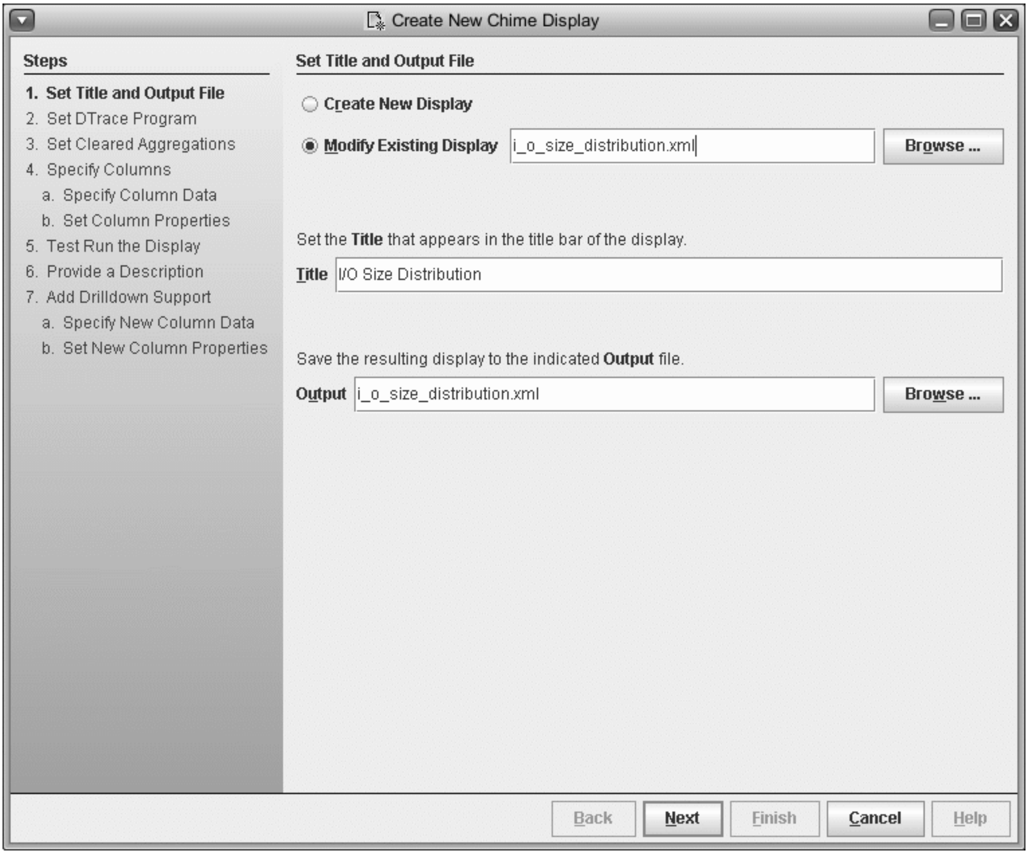


Figure 13-2 Customizing and enhancing Chime by adding new "displays"

scripts (see Figure 13-2). Chime is distributed with starter scripts, many of which are from the DTraceToolkit.[2]

Examples

Chime can be used at the command line as a replacement for DTrace, for one-liners and scripts that populate aggregations. For example, the following one-liner can be executed at the command line:

2. See *http://blogs.sun.com/tomee/entry/chime_and_the_dtracetoolkit* and *http://blogs.sun.com/tomee/entry/chime_and_the_dtracetoolkit_part*.

```
solaris# dtrace -n 'sysinfo:::readch { @bytes[execname] = sum(arg0); }'
solaris# /opt/OSOL0chime/bin/chime -n 'sysinfo:::readch { @bytes[execname] =
sum(arg0); }'
```

or via Chime (see Figure 13-3).

Newer features of Chime include automatic drilldown analysis;[3] in Figure 13-4, this is demonstrated for the gnome-panel process, drilling down on system calls by function.

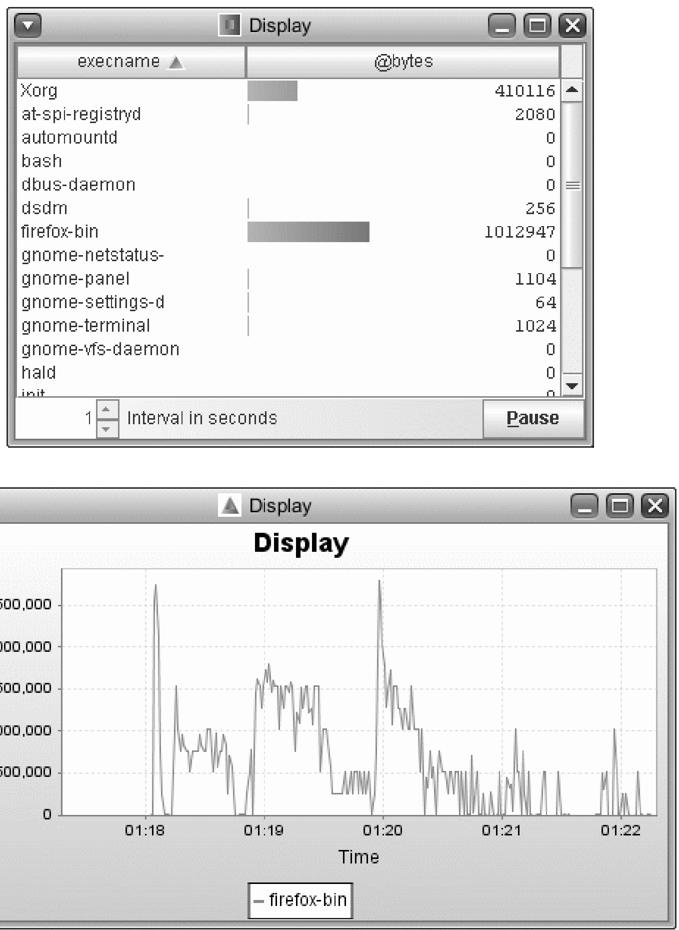


Figure 13-3 Chime displaying read bytes by process, and for firefox-bin only

3. *http://blogs.sun.com/tomee/entry/chime_automatic_drilldown*

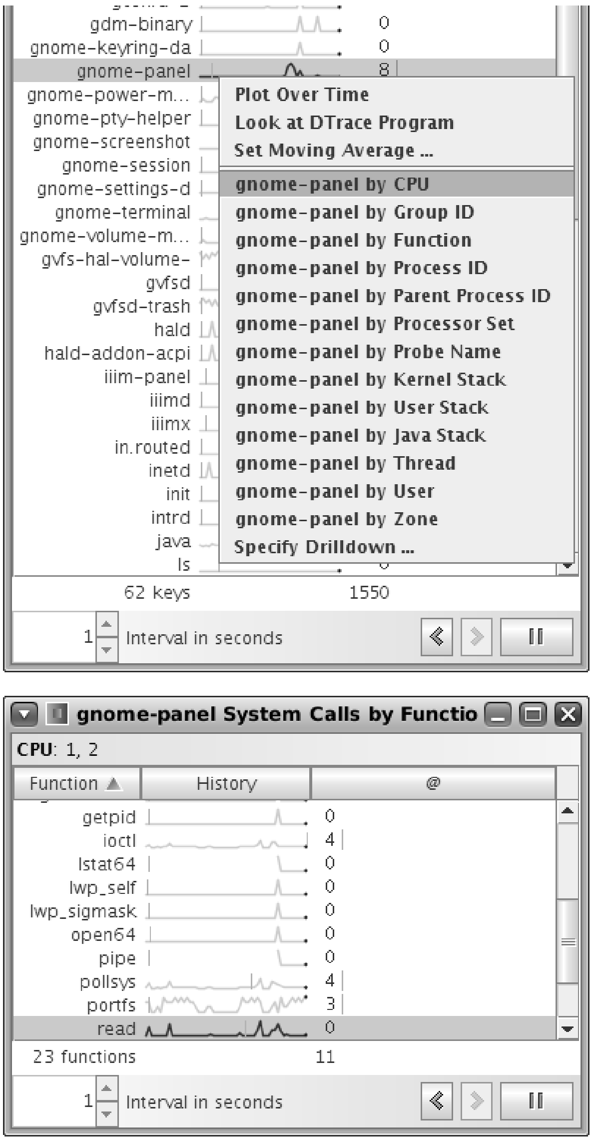


Figure 13-4 Chime drill-down analysis

DTrace GUI Plug-in for NetBeans and Sun Studio

The DTrace GUI plug-in is a graphical interface for DTrace that can be installed into the Sun Studio IDE, Oracle Solaris Studio IDE, and currently available releases of the NetBeans IDE.[4]

It was written by Nasser Nouri and is based on Chime and the DTraceToolkit (the scripts from which are included under a `DTraceScripts` subdirectory).

Location

Information on the DTrace GUI plug-in, along with download and install instructions, can be found at *http://wiki.netbeans.org/DTrace*.

Examples

Figure 13-5 shows selecting, editing, and executing `vmbypid.d` in the NetBeans IDE.

DLight, Oracle Solaris Studio 12.2

Oracle Solaris Studio 12.2 can use DTrace for performance analysis. A kernel profiler tool called `er_kernel` is included, which uses DTrace to sample kernel stack traces and produces a report.

Oracle Solaris Studio 12.2 includes DLight, a standalone interactive observability tool that analyzes data from multiple DTrace scripts in a synchronized fashion to trace a runtime problem in an application to its root cause. DLight can analyze an executable or a running process. It includes five profiling tools for C, C++, and Fortran programs.

> **Thread Microstates**: This provides an overview of the program's threads as they enter various execution states during the program's run. The Solaris microstate accounting feature uses the DTrace facility to provide fine-grained information about the state of each thread as it enters and exits ten different execution states.

4. NetBeans is available as a standalone IDE. Sun Studio and Oracle Solaris Studio, when installed, include NetBeans.

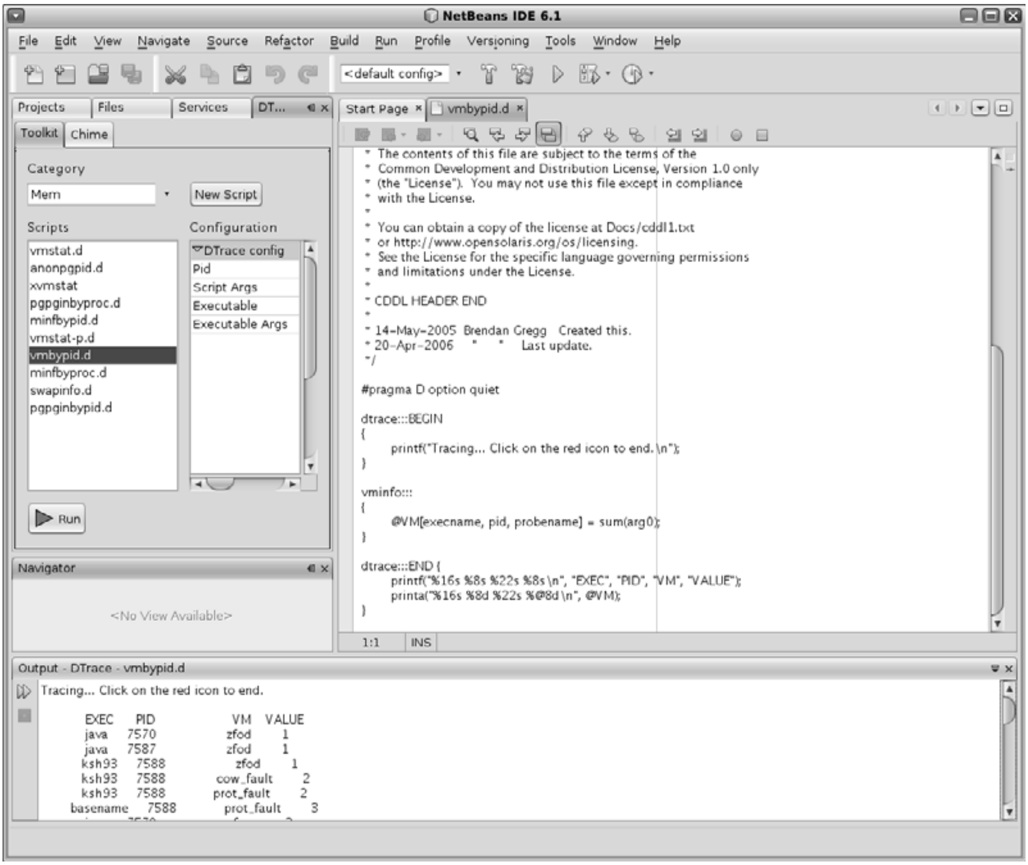


Figure 13-5 NetBeans interface for selecting and running scripts

CPU Usage: This provides the percentage of the total CPU time used by the program during its run.

Memory Usage: This shows the way the program's memory heap changes over time. This tool identifies memory leaks, which are points in the program where memory that is no longer needed fails to be released. These leaks can lead to increased memory consumption and even cause a program to run out of usable memory.

Thread Usage: This provides the number of threads in use by the program, and any moments where a thead has to wait to get a lock in order to proceed with its task. This data is useful for multithreaded applications, which must perform thread synchronization in order to avoid expensive wait times.

I/O Usage: This provides an overview of the program's read and write activity during the run.

Figure 13-6 provides an illustration of the DLight profiling tools.

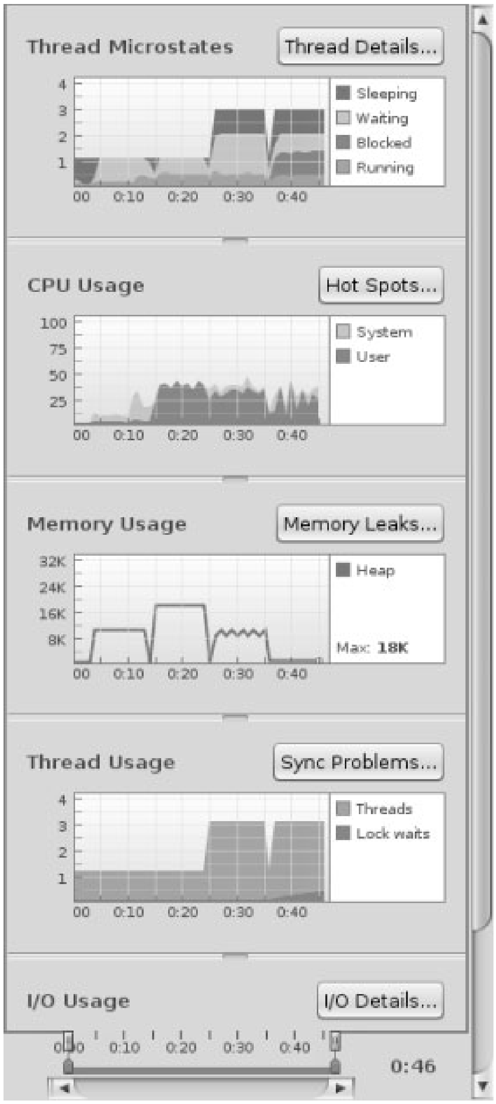


Figure 13-6 DLight profiling tools

Locations

Documentation for these can be found at the following locations:

> *http://docs.sun.com/app/docs/doc/821-0304*: The Performance Analyzer manual of the Oracle Sun Studio collection
>
> *http://docs.sun.com/app/docs/doc/821-2126*: Oracle Solaris Studio 12.2 DLight Tutorial

Examples

Figures 13-7 through 13-10 provide examples of DLight screens and output.

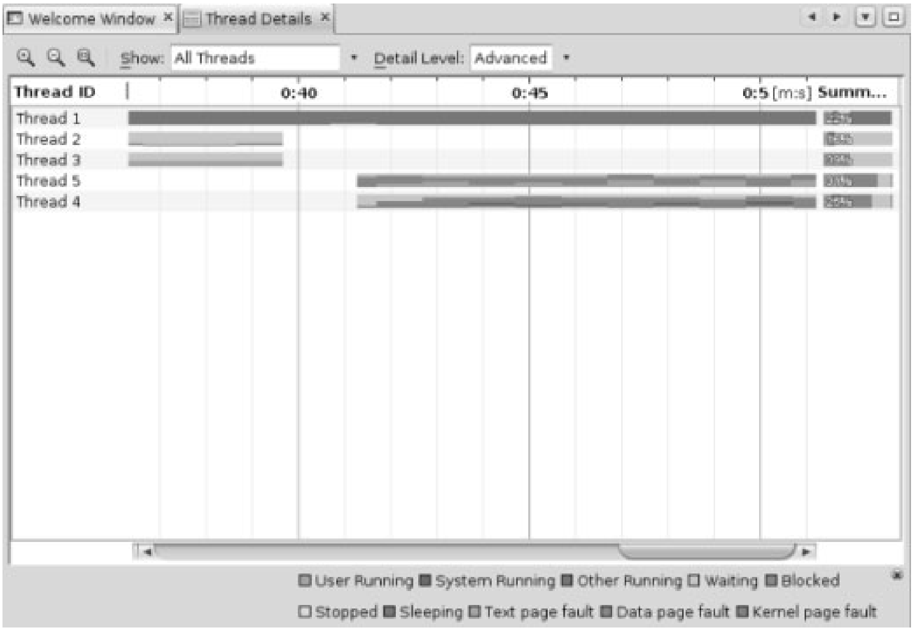


Figure 13-7 DLight Thread Details window

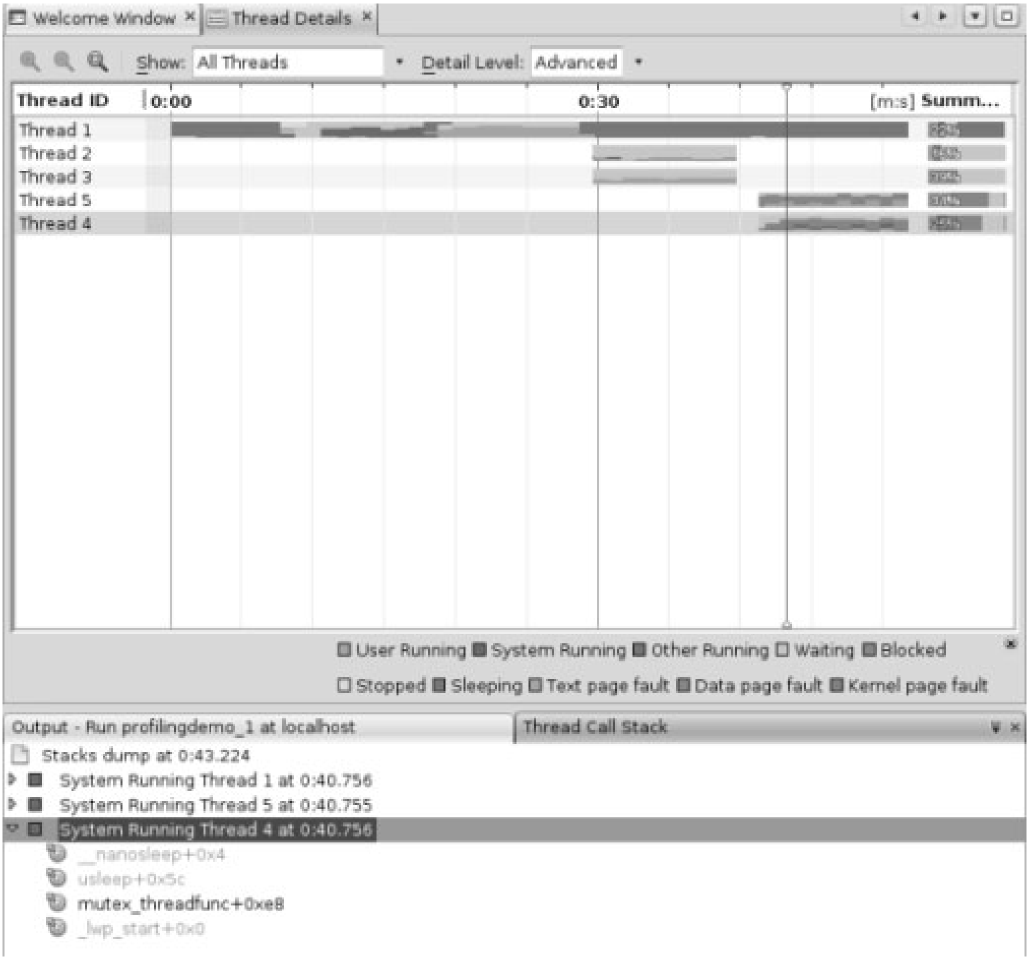


Figure 13-8 DLight thread call stack

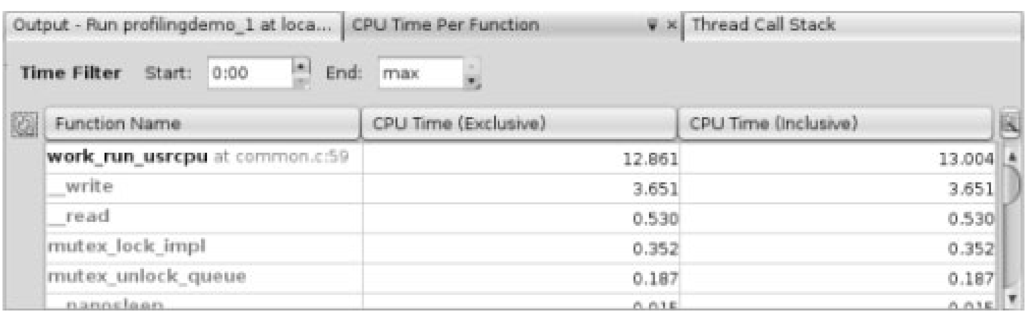


Figure 13-9 DLight CPU usage details

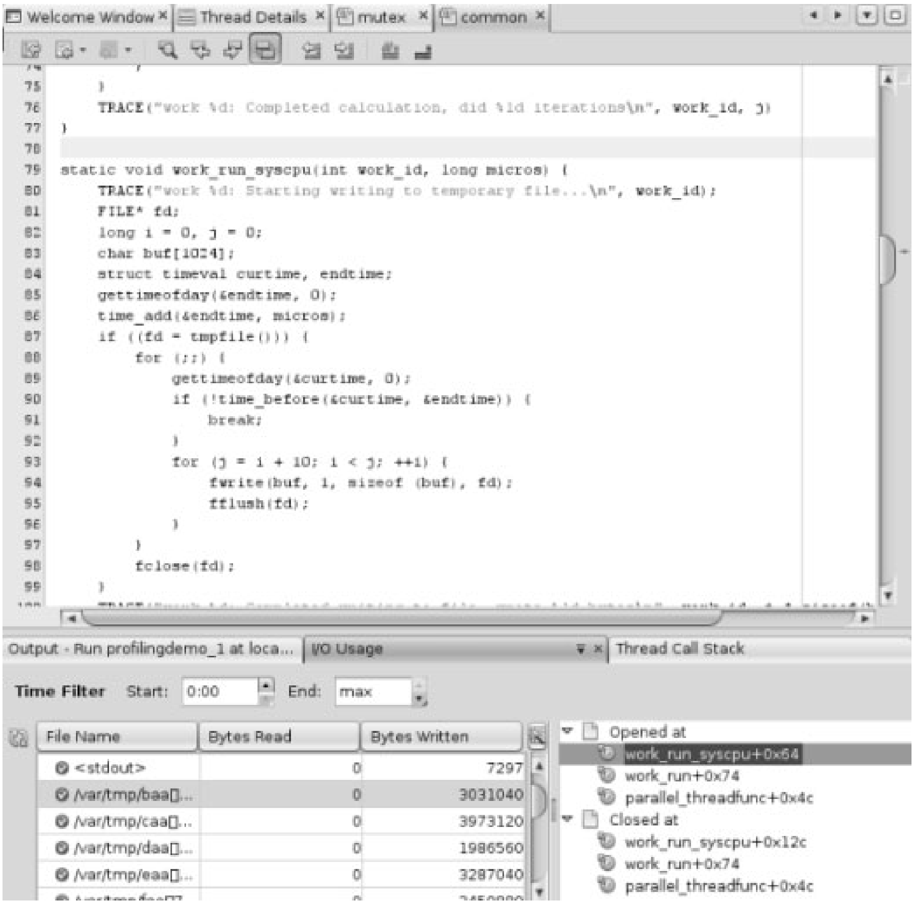


Figure 13-10 DLight I/O usage details

Mac OS X Instruments

Mac OS X Instruments is a graphical analysis tool for tracing and profiling code. It can use DTrace to fetch data, and according to the Instruments User Guide, much of Instruments is now based on DTrace:

> DTrace is a dynamic tracing facility originally created by Sun and ported to Mac OS X v10.5. Because DTrace taps into the operating system kernel, you have access to low-level operation about the kernel itself and about the user processes running on your computer. Many of the built-in instruments are already based on DTrace. And even though DTrace is itself a very powerful and complex tool, Instruments provides a simple interface that gives you access to the power of DTrace without the complexity.

Locations

Instruments is part of the Mac OS X Xcode developer tools, which can be downloaded from *http://developer.apple.com/technologies/xcode.html*.

The user guide is at *http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide*.

Of particular interest is the chapter "Creating Custom Instruments with DTrace."

Instruments is part of the Xcode Tools developer suite, which can be downloaded from the members area of the Apple Developer Connection Web site (free registration required) at *http://connect.apple.com/*.

Examples

Figure 13-11 shows an example of output from Instruments.

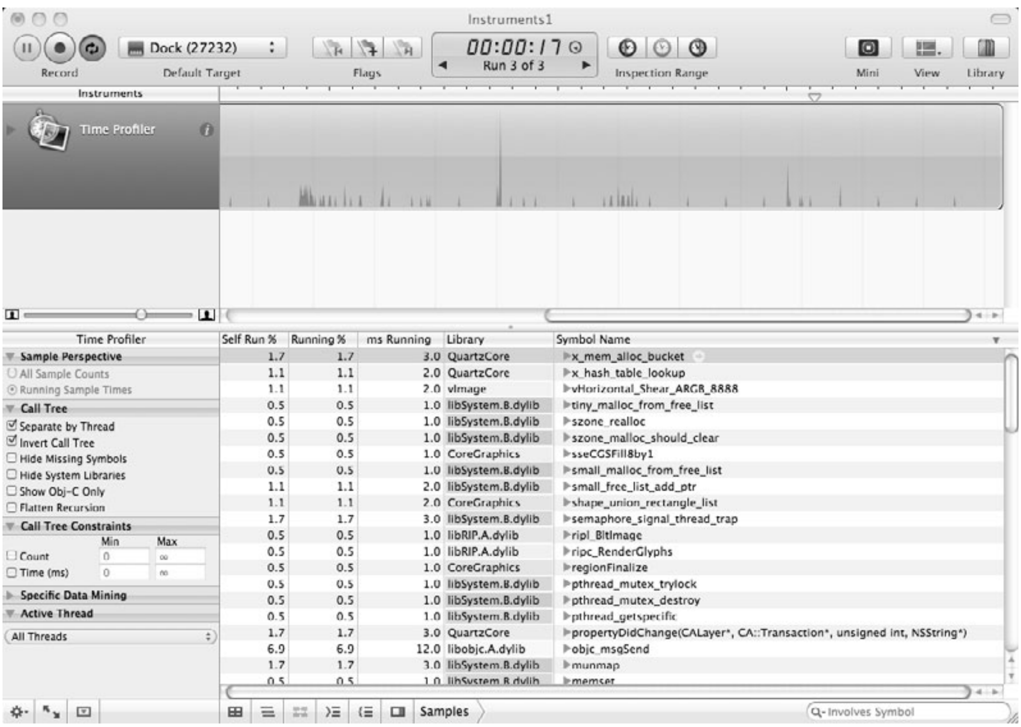


Figure 13-11 Mac OS X Instruments example

Analytics

By now you have learned enough about DTrace to understand the raw analytical power that it offers to a skilled practitioner—emphasis on *skilled*. And it takes time, ability, and experimentation to become a DTrace expert. What if there was an easier way for nonexperts to use the power of DTrace to diagnose complex systems in real time?

Some of the finest minds at Sun Microsystems were applied to this question. The result was Analytics,[5] a DTrace-based graphical analysis tool released in late 2008 as part of the Sun ZFS storage appliance (the 7000 series).

Analytics enables you to construct complex DTrace queries via a simple point-and-click graphical interface, designed to promote drill-down analysis without knowledge of DTrace or operating system internals. It uses visualizations to present the resulting data in ways that add value, aiding interpretation with line plots, stacked plots, heat maps, hierarchy views, and pie charts—whatever is most effective for the data type presented. A system can be analyzed in real time, with data retained at a one-second granularity for postevent analysis.

This chapter introduces Analytics as a case study in using DTrace via a graphical environment and illustrates how visualizations can enhance the data that DTrace makes available. This material is based on a presentation[6] that Bryan Cantrill delivered for CEC (Sun's Customer Engineering Conference) in 2008, when Analytics was released as part of the Sun 7000 storage appliance. A VirtualBox simulator version[7] of the storage appliance is currently available as a free download, so you can try Analytics without its associated storage hardware.

The Problem

Historically, storage administrators have had very little insight into the nature of performance; essential questions like "What am I serving and to whom?" or "And how long is that taking?" were largely unanswerable.

The problem is made acute by the central role of storage in information infrastructure—it has become very easy for applications to "blame storage." It has therefore become the storage administrator's problem to exonerate the storage infrastructure. However, with the limited tool set available until now, this has been excruciating to impossible.

5. Analytics was also primarily created and developed by Bryan Cantrill, coinventor of DTrace.
6. *http://blogs.sun.com/fishworks/resource/CEC08/fishworks_analytics.pdf*
7. *http://blogs.sun.com/fishworks*

Those best positioned to shed some light on storage systems are those with the greatest expertise in those systems: the vendors. But vendors seem to have the same solutions for every performance problem: Buy more/faster/bigger disks/systems. This costs the customer a boatload—and doesn't necessarily solve the problem!

Solving the Problem

How can a storage administrator understand what's really going on in the storage infrastructure? An effective storage observability solution needs the following:

- A way to understand storage systems not in terms of their implementation but rather in terms of their abstractions
- To be able to quickly differentiate between problems of load and problems of architecture
- To allow you to quickly progress through the diagnostic cycle: from hypothesis to data and then to new hypothesis and new data
- To be graphical—it should harness the power of the visual cortex
- To be real-time—it needs to be able to react quickly to changing conditions

To best understand these, they are explained as follows in terms of the problem only—not in terms of any possible solution (DTrace).

Implementation vs. Abstraction

Understanding the system's implementation—network, CPU, DRAM, disks—is useful only when correlated to the system's abstractions. For a storage appliance, the abstractions are at the storage protocol level, for example:

- NFS operations from clients on files
- CIFS operations from clients on files
- iSCSI operations from clients on volumes

These abstractions describe the load applied to the appliance. The interfaces to a storage appliance are the protocols it supports: NFS, CIFS, iSCSI, and so on. The applied load is expressed in these terms. The load is *not* CPU cycles, disk IOPS, or network packets—those are implementation details that can occur as a *result of* load.

Load vs. Architecture

Performance is the result of a given load (the work to be done) on a given architecture (the means to perform that work). We shouldn't assume that poor performance is the result of inadequate architecture; it may be because of unnecessarily high load. The system cannot automatically know whether the load or the architecture (or both) is ultimately at fault; to determine that, both must be observable by the administrator.

Diagnostic Cycle

The diagnostic cycle is the progression from hypothesis through instrumentation to data gathering to a new hypothesis:

hypothesis → instrumentation → data → hypothesis

Enabling the diagnostic cycle has implications for any solution to the storage observability problem: The system must be highly interactive to allow new data to be quickly transformed into a new hypothesis, and the system must allow ad hoc instrumentation to be created, specific to the data that motivates it.

Visualizations

The human brain has evolved an extraordinary ability to visually recognize patterns. Tables of data are often ineffective. We must be able to visually represent data to perceive subtle patterns. In the case of Analytics, this does not mean merely "adding a GUI" or bolting on a third-party graphing package but rather rethinking how we visualize performance. Visualization must be treated as a first-class aspect of the storage observability problem.

Real Time

The storage administrator needs to be able to interact with the system in real time to understand the dynamics of the system. They also need to be able to understand the system at a fine temporal granularity (for example, one second); coarser granularity only clouds data and delays response.

Toward a Solution

DTrace is well suited to be the foundation for a storage observability solution. It cuts through implementation to get to the semantics of the system and separates architectural limitations from load-based pathologies.

However, DTrace is only a foundation. The real win for the user is to create a powerful and usable system that makes it easy to generate DTrace queries—an abstraction layer above the programmatic interface. We also need a means to visualize the data and the ability to (efficiently!) store historical data.

Appliance Analytics

Figure 13-12 shows a screen from Analytics.

Analytics is a DTrace-based facility that allows storage administrators to ask questions phrased in terms of storage abstractions.

> "What clients are making NFS requests?"
>
> "What CIFS files are being accessed?"

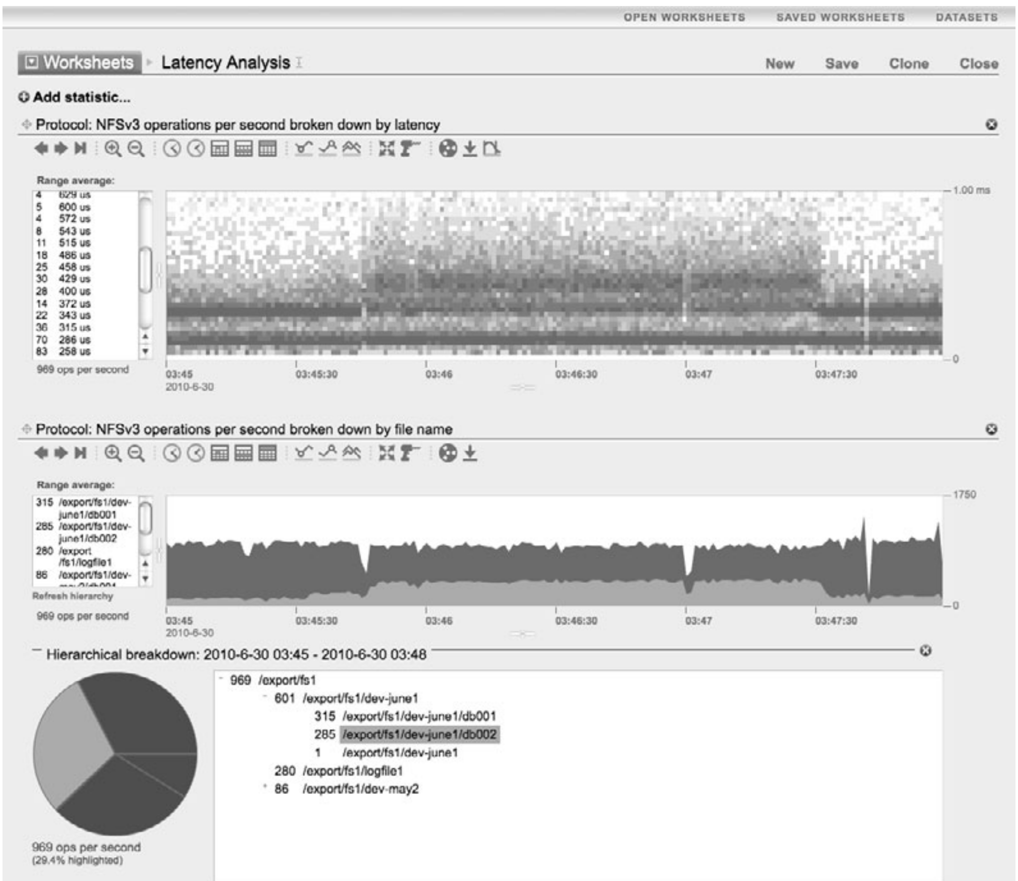


Figure 13-12 Analytics screen

"What LUNs are currently being written to?"

"How long are CIFS operations taking?"

The data to answer these questions is represented visually, with the browser as vector. All data is available at a per-second granularity, can be viewed in real time, and can be optionally recorded for historical analysis.

Analytics can also answer much more complex queries that can be formulated through an intuitive visual interface.

"What files are being accessed by the client `kiowa`?"

"What is the read/write mix for the file `usertab.dbf` when accessed from client `deimos`?"

"For writes to the file `usertab.dbf` from the client `deimos` taking longer than 1.5 milliseconds, what is the file offset?"

These queries can be formulated in real time based on past data. The data from these queries can themselves be optionally recorded, and the resulting data can become the foundations for more detailed queries.

The sections that follow explain key components of Analytics.

Statistics

Analytics displays and manipulates "statistics." A statistic can be a "raw statistic"—a single scalar recorded over time (for example, "NFSv3 operations per second"). Statistics can also be broken down into constituent elements (for example, "NFSv3 operations per second broken down by client").

Statistics are examined in Analytics by clicking an Add Statistic... button. This displays a menu of statistics, each of which may have suboptions to display different breakdown dimensions for that statistic, as shown in Figure 13-13.

The available statistics are designed for effective observability of storage load and architecture, with only the most important of these statistics displayed by default; the full set of statistics is displayed only when an "advanced analytics" option is enabled.

Once a statistic is selected, a new panel is added to the display, containing a graph of the statistic, updated in real time (see Figure 13-14).

Time is on the x-axis, moving from right to left, and value is on the y-axis. The average over the visible time range (entire x-axis) is displayed to the left of the graph (Figure 13-14 shows 1147 NFSv3 ops/sec, which is the average for the 60 seconds displayed). To get the value of a statistic at a particular time, click that time in the graph to display the value.

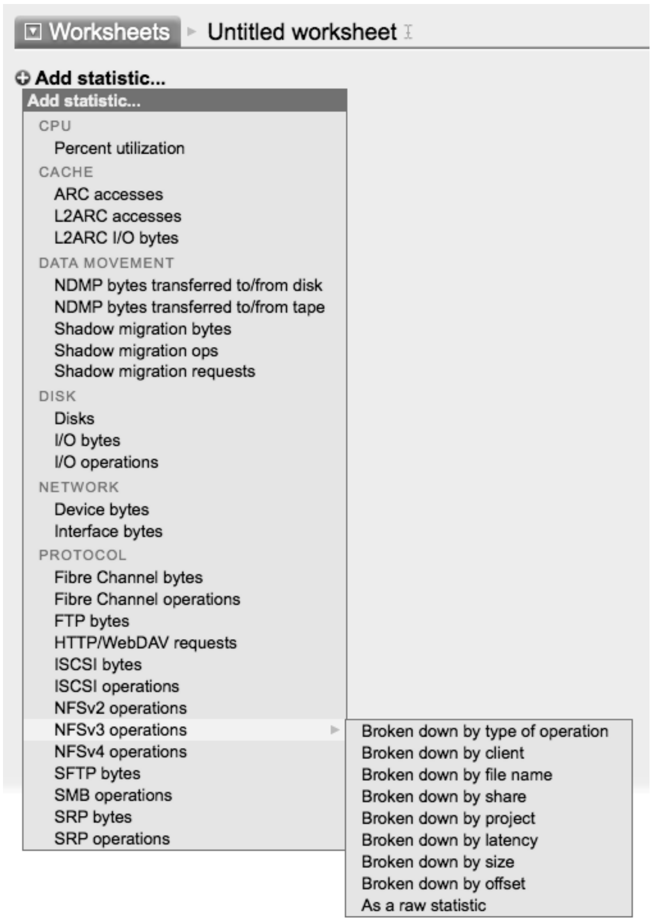


Figure 13-13 Adding a statistic

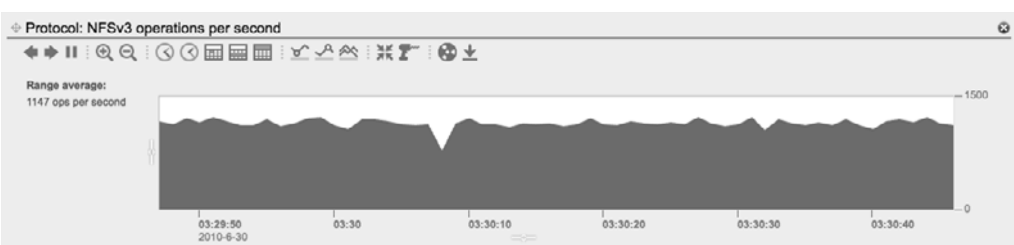


Figure 13-14 Raw statistic

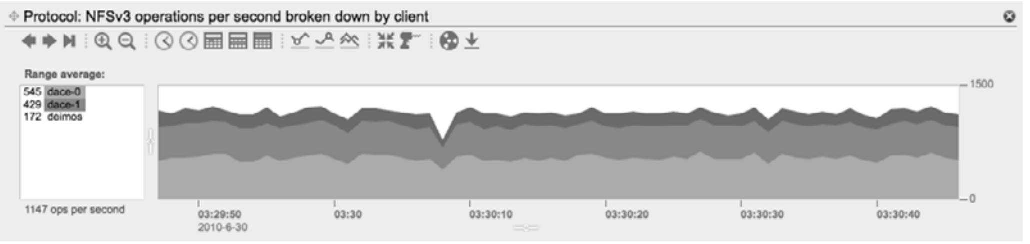


Figure 13-15 Breakdown statistic

Breakdowns

Many of the statistics can be broken down in a variety of ways, producing "breakdown" statistics. Figure 13-13 shows the available breakdowns for the "NFSv3 operations per second" statistic, which are type of operation, file name, client, share, project, latency, size, and offset.

For breakdown statistics, the area to the left of the graph contains a breakdown table showing average value of each element. One or more breakdown elements can be highlighted by clicking and Shift+clicking them in the table (see Figure 13-15).

The table lists the top ten elements over the displayed time period; if more elements are available, an ellipsis (...) will appear as last element in table, which can be clicked to reveal more elements.

Hierarchical Breakdowns

Some of the breakdowns have a hierarchy of elements, such as files in a directory tree or disk devices in a device tree (which includes host bus adaptor cards and disk enclosures). Analytics can visualize files and devices hierarchically by clicking "Show hierarchy" under the breakdown table. The hierarchy can be expanded by clicking plus (+) buttons, as shown in Figure 13-16.

The pie chart provides visual comparison of selected breakdowns. The wedges can also be clicked to toggle highlighting. Figure 13-16 shows two files and one directory highlighted.

Heat Maps

For some statistics, such as operation latency, size, offset, and so on—a scalar is not sufficiently expressive. A scalar average can be highly misleading, and zero-valued data must be distinguished from no data. For these operations, Analytics allows the distribution of data over time to be examined using quantized breakdowns. These consist of time on the x-axis, values on the x-axis, and a heat map (a color-coded histogram) per sample (see Figure 13-17).

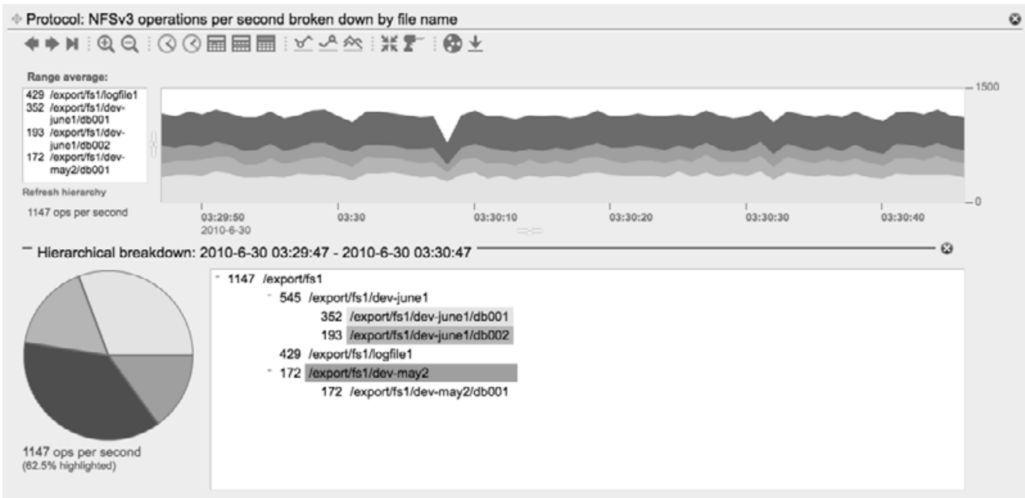


Figure 13-16 Hierarchical breakdowns

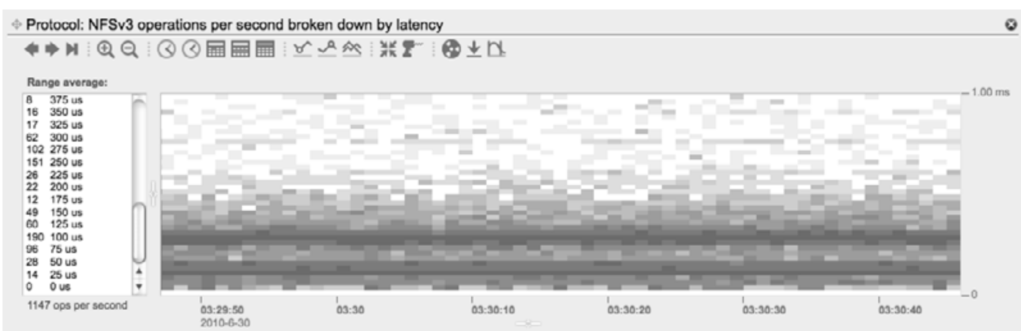


Figure 13-17 Heat map

The color of each time/latency pixel represents the number of operations at that time and latency range. The darker the color, the more operations occurred. A false color palette is also applied to highlight subtle details.

For heat maps of latency, occasional high latency (*outliers*) will be easily identified as lone pixels at the top of the heat map. This aids identification of outliers, which for latency can represent performance issues. However, it can also compress the bulk of the lower latency data on the y-axis, making examination of the normal latency ranges difficult; to manage this, Analytics has an outliers elimination button to control whether outliers are included in the display and are allowing the y-axis to be zoomed to the range of interest.

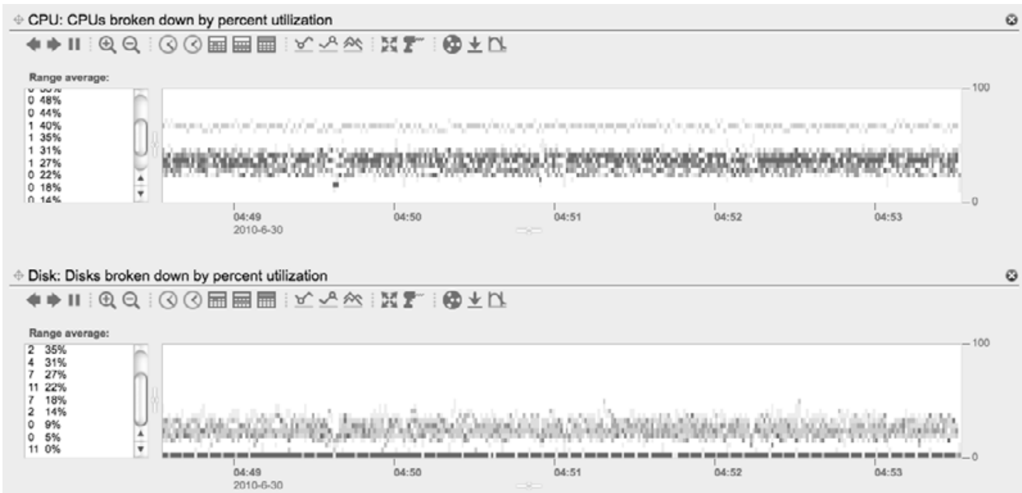


Figure 13-18 Utilization heat map

The heat map shown in Figure 13-17 shows two levels of latency, represented as the dark horizontal lines. This detail would remain unknown if a line plot of average latency was examined instead.

Another breakdown that uses the heat map visualization is utilization, as shown in Figure 13-18.

Instead of showing average utilization across all CPUs or disks, a heat map with utilization on the y-axis is used, where the color for each time/utilization range represents the number of devices at that utilization level. This allows various problems to be identified: high utilization, poor scaling across components, and single bad components (for example, a disk that is stuck on 100 percent utilization or a software thread that is pinning a single CPU at 100 percent utilization). Figure 13-18 shows that a few CPUs are at a higher rate of utilization than the rest, and several disks are completely unutilized (spares).

Drill-Downs

Ad hoc queries are formed by drilling down on a particular element in a breakdown statistic. The possible drill-downs for an element are shown when right-clicking the element, as shown in Figure 13-19.

If the drill-down statistic has further breakdowns, they can be selected to continue drilling down further. For example, starting with the raw statistic for "NFSv3 operations per second" during load, the following four drill-downs were performed by continuing to drill down on elements:

Figure 13-19 Drilling down with the mouse

1. NFSv3 operations per second, broken down by type
2. NFSv3 operations per second of type read, broken down by client
3. NFSv3 operations per second of type read for client `dace-1`, broken down by file name
4. NFSv3 operations per second of type read for client `dace-1`, for file `/export/fs1/logfile1` broken down by offset

The last two are shown in Figure 13-20.

Behind the scenes, a DTrace script is being dynamically created from these mouse clicks. Usually—depending on the statistic and breakdown—Analytics will

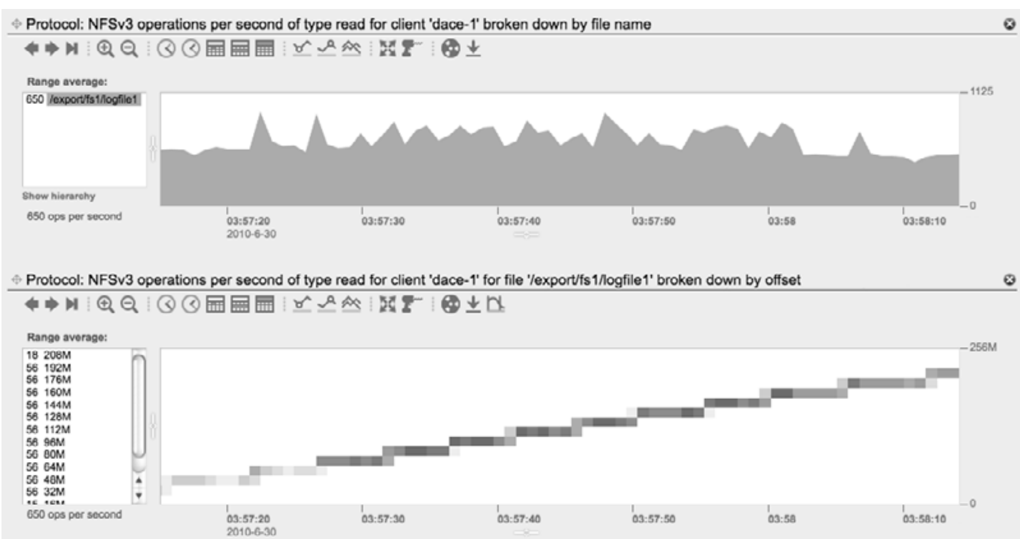


Figure 13-20 Complex drill-downs

seamlessly use other sources for statistics if available and appropriate, such as Kstat (kernel statistics).

Controls

Above each graph is a set of controls for navigation of the statistics, as shown in Figure 13-21.

There are 18 buttons in the control bar, each an icon to represent the action it performs. Table 13-2 summarizes the function of these controls.

These are included here to further explain the Appliance interface; for a full description of the functionality, see the product documents.[8]

Worksheets

Statistics are viewed on a worksheet, which can be named and saved. This allows custom observability tools to be constructed as worksheets containing all the statistics of interest.

Figure 13-21 Analytics control bar

Table 13-2 Analytics Control Descriptions

Button	Function
1, 2, 3	Moving: left, right, pause
4, 5	Zooming: in, out (x-axis)
6, 7, 8, 9, 10	Time scale: minute, hour, day, week, month
11, 12	Find value: minimum, maximum
13	Toggle graph type: multi line / stacked
14	Synchronizing graphs
15	Drill down
16	Save this statistic as a dataset
17	Export visible statistic as CSV
18	Outlier elimination (y-axis zoom)

8. These are both bundled with the appliance under the HELP wiki and in the Administration Guide, currently at *http://wikis.sun.com/display/Fishworks/Documentation*.

By pausing a worksheet on a particular time and then saving it, you can easily return to that time by opening the worksheet, which opens up all the statistics that were being observed. This can be useful when examining an event such as a performance issue, where a worksheet can be opened by other staff who will be taken straight to the time and statistics for that event.

An entire worksheet can also be downloaded as a bundle for analysis on remote systems. The bundle includes all the statistics in the worksheet, bounded by the selected time range (x-axis) in the worksheet. It allows zooming in to one-second granularity. These bundles are used during support, where a customer can create a worksheet that spans an interesting event, and send a support engineer a bundle containing all the data for remote analysis.

Datasets

Analytics allows statistics to be archived for historical analysis: These archived statistics are called *datasets*, and they contain all data at a one-second granularity. Recording them means that multiple DTrace enablings (one for each statistic) are running continually.

To manage the size of data on disk, the administrator can check the size of these datasets on the Analytics > Datasets screen and destroy or suspend any that are too large. Datasets that can become particularly large include the by-file breakdowns when serving thousands of files, since each second contains a list of filenames that were accessed, along with their operation/sec counts. Heat maps can become large, too, since the color of the pixel adds a third dimension. All of this data is compressed on disk, by placing it on an Oracle ZFS share where compression is enabled.

The CPU overhead of recording these datasets has already been minimized by the design of DTrace and is typically negligible. It is relative to the frequency of events, and so is most noticeable when tracing network packets (which can reach hundreds of thousands of packets per second). Times when this overhead can be a factor include performing benchmarks that drive the system to its maximum, where every last percentage point of performance matters. In those cases, the more CPU expensive datasets can be suspended during the benchmark.

See Also

This section briefly showed how Analytics visualizes the data that DTrace makes available, as a DTrace visual interface case study. For more information about this topic, see the following:

> "Visualizing System Latency" by Brendan Gregg, Communications of the ACM, July 2010

"Analytics in the Sun 7000 Series" by Bryan Cantrill and Brendan Gregg[9]

Chapter 6, "Analytics," of the Oracle Sun ZFS Storage 7000 Administration Guide[10]

Summary

This chapter demonstrated various tools built atop of DTrace, including the DTraceToolkit, a large collection of scripts; and Analytics, a sophisticated GUI analyzer that facilitates drill-down analysis and adds value to the data DTrace makes available.

9. *http://blogs.sun.com/bmc/resource/cec_analytics.pdf*

10. *http://wikis.sun.com/display/FishWorks/Documentation*

This page intentionally left blank

14

Tips and Tricks

DTrace is an extremely powerful and versatile tool, designed to provide observability into systems that are inherently very complex. As with any other technology, there is a learning curve, and as one gains experience with DTrace, one discovers interesting ways to dig deeper and learn more about the systems and software being analyzed and leverage the power of DTrace more effectively. Many have been using DTrace extensively for many years and from that experience have gained insight that can be shared with those at various stages of the DTrace learning curve. In this chapter, we offer some tips and tricks to help you learn and apply DTrace effectively based on our experiences and the experience of many others.[1]

Tip 1: Known Workloads

Before using DTrace to examine an unknown workload, consider generating a known workload to examine as a controlled experiment. This should help you write correct DTrace scripts much more quickly, because you can check the script output vs. the known workload input. If there are systemic nuances that complicate analysis, they will be easier to grasp in the context of a known workload than of one that is unknown.

1. *http://dtrace.org/blogs/bmc/2005/06/20/yet-more-blog-sifting/* contains links to some very interesting and educational blogs on having used DTrace to solve a specific problem.

As an example, let's say I'd like to analyze NFS reads on a busy Solaris NFS server. Rather than writing scripts to analyze the production workload, of which I don't yet have a clear understanding, my task will be easier if I first analyze a known workload. To create one, use the dd command:

```
client# mount 192.168.56.3:/export/fs1 /mnt
client# dd if=/mnt/100m of=/dev/null bs=1024 count=5
5+0 records in
5+0 records out
5120 bytes transferred in 0.006986 secs (732905 bytes/sec)
```

The client is Mac OS X, which mounts the share using NFSv3. The arguments to dd shown earlier create a known workload of five 1KB reads—a simple workload to start with.

I then used DTrace on the NFS server to trace these reads. Since the NFS server is busy processing other client requests, I used a predicate to match only reads from our client, 192.168.56.1. The filename and size of each read were printed:

```
server# dtrace -n 'nfsv3:::op-read-start /args[0]->ci_remote == "192.168.56.1"/ { prin
tf("%s read %d bytes", args[1]->noi_curpath, args[2]->count); }'
dtrace: description 'nfsv3:::op-read-start ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
  0  90345          rfs3_read:op-read-start /export/fs1/100m read 32768 bytes
^C
```

On the NFS server, there were 17 32KB reads to the file. This doesn't match at all what dd requested. To double-check, I repeated the dd command while DTrace was tracing:

```
client# dd if=/mnt/100m of=/dev/null bs=1024 count=5
5+0 records in
5+0 records out
5120 bytes transferred in 0.000310 secs (16519105 bytes/sec)
```

```
server# dtrace -n 'nfsv3:::op-read-start /args[0]->ci_remote == "192.168.56.1"/ { prin
tf("%s read %d bytes", args[1]->noi_curpath, args[2]->count); }'
dtrace: description 'nfsv3:::op-read-start ' matched 1 probe
^C
```

This time *no* NFS reads were seen on the server, despite `dd` successfully completing five reads.

If until now everything has behaved as you expected it, then you haven't wasted much time confirming what you already know (and you know your NFS pretty well).

If, instead, you are surprised by the mismatch, then you have made an important discovery: Something went wrong with the workload, with the DTrace one-liner, or with your understanding of NFS. At this point, you don't know which, but you can perform a simple experiment to figure it out.

Note

The issue is that `dd` is performing reads on the client, and those reads are being processed by the client NFS driver. In the first case, the reads triggered the client NFS driver to perform read-ahead. Read-ahead is where more data is fetched than was requested to prewarm the DRAM-based client cache, in an effort to improve performance. In the second case, no NFS reads were requested since the client satisfied the reads from its own client cache.

Discoveries like these are common when using DTrace. By generating a simple known workload and then tracing it, you will quickly iron out any issues before encountering them in the unknown, and typically more complex, production workload. Also, the use of available benchmarking utilities makes it relatively easy to generate a known load that serves two purposes—developing your DTrace and understanding the data values returned, and verifying that the underlying subsystem being measured can perform as expected. Utilities such as `uperf` or `ttcp` for network loading and `filebench` for file system I/O loads are readily available and easy to use.

Tip 2: Write Target Software

The previous tip was to start with a simple workload. Here we start with a simple target.

Production applications can get complicated, and it can be difficult to learn DTrace while at the same time learning the internals of a complex system. If components of the production environment can be reproduced separately on a lab system, they can be separately DTraced and understood before tackling them in the context of the production environment. This will typically involve writing software (which often reveals implementation details that are useful to know).

Apart from replicating parts of the production application, you can also replicate specific issues in order to test analysis techniques and fine-tune DTrace scripts. This could include small programs that exhibit hot code paths, lock contention, disk I/O, and network I/O.

As a simple example: An application is hot on-CPU, and it is important to quantify which functions are responsible. To test whether profiling the on-CPU user-land function might be an effective technique, a C program was written (`hotcpu`) to generate a known workload:

```
int i, j;

void func_alpha()
{
        for (i = 0, j = 1; i < 1 * 1000000; i++) { j++; }
}

void func_beta()
{
        for (i = 0, j = 1; i < 10 * 1000000; i++) { j++; }
}

int main(int argc, char *argv[])
{
        while (1) {
                func_alpha();
                func_beta();
        }
        return (0);
}
```

This should cause `func_beta()` to be on-CPU ten times longer than `func_alpha()`. This program was run and the user-land function was sampled at 1001 Hertz:

```
# dtrace -n 'profile-1001 /execname == "hotcpu"/ { @[ufunc(arg1)] = count(); }'
dtrace: description 'profile-1001 ' matched 1 probe
^C

  hotcpu`func_alpha                                                290
  hotcpu`func_beta                                                2905
```

The output matches expectations: `func_beta()` was on-CPU during 2,905 samples, which is ten times `func_alpha()` at 290 samples. This approach can now be applied to the target application with a greater degree of confidence.

Although this is only a simple example, the concept can be applied to much more complex issues. Imagine writing a short program to exhibit a known rate of adaptive mutex lock contention and using this program to develop and fine-tune lock analysis scripts.

Tip 3: Use grep to Search for Probes

This may be obvious, but it's worth mentioning: `grep(1)` can filter probe lists. Some providers, such as fbt, make tens of thousands of probes available. A quick way to find interesting probes from such a list can be to search for likely keywords using `grep`. For example, searching for *keyboard* shows related probes in the fbt provider on Solaris:

```
# dtrace -ln fbt:::entry | grep keyboard
71069        fbt   consconfig_dacf            plat_stdin_is_keyboard entry
72619        fbt           kbtrans      kbtrans_streams_set_keyboard entry
73083        fbt            kb8042           kb8042_send_to_keyboard entry
```

This has also matched modules that may be of interest, `kbtrans` and `kb8042`. DTrace can perform simple searches like this using wildcards (`*`) as part of the probe name (for this example, running `dtrace -l 'fbt::*keyboard*:entry'`), but because it supports regular expressions, `grep` can apply more powerful filters.

Tip 4: Frequency Count

Another way to find probes of interest is to apply a known workload (see tip 1) and frequency count probes to see which fire at a rate similar to the workload.

For example, let's say I wanted to investigate how ZFS processed `mkdir` on Solaris, but I don't know which probe to start tracing in the ZFS kernel module. As a known workload, I run a shell script that runs `mkdir` 23 times from a ZFS directory, while frequency counting all zfs functions:

```
window1# ./run_mkdir_23_times.ksh
window2# dtrace -n 'fbt:zfs::entry { @[probefunc] = count(); }'
dtrace: description 'fbt:zfs::entry ' matched 1751 probes
^C

  bplist_vacate                                                   1
  dbuf_fill_done                                                  1
  dbuf_noread                                                     1
  dmu_buf_will_fill                                               1
  dmu_free_range                                                  1
[... output truncated ...]
  metaslab_compare                                               18
```

continues

```
  dmu_zfetch_stream_remove                                          19
  zfs_readdir                                                       22
  mzap_create_impl                                                  23
  zap_create_norm                                                   23
  zfs_mkdir                                                         23
  arc_free                                                          24
  dsl_dataset_block_born                                            24
  dsl_dataset_block_kill                                            24
[... output truncated ...]
  zio_wait_for_children                                          29891
  dbuf_hash                                                      30199
  dbuf_rele                                                      31273
  dbuf_read                                                      34549
  propname_match                                                 74874
  zprop_name_to_prop_cb                                          74874
```

While the output was many pages long (containing more than 600 functions), only three fired 23 times, matching the known workload. These included `zfs_mkdir()`, which sounds like the best function from which to begin this investigation (I could also have found this with Tip 3).

Tip 5: Time Stamp Column, Postsort

On multi-CPU systems, the output of DTrace can become slightly shuffled because of the way DTrace collects buffer data from each CPU in turn and prints it at a default rate of 1 Hertz (configurable using the `switchrate` tunable). When it's important to analyze the output in the correct order, print a time stamp column and postsort. Some DTrace-based scripts such as `iosnoop` have an option to do this (`iosnoop -t`), which could then be postprocessed by the command-line `sort(1)` utility (using a numeric sort: `sort -n`).

The following demonstrates the issue:

```
# dtrace -n 'profile:::profile-3hz { trace(timestamp); }'
dtrace: description 'profile-3hz ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  41241                     :profile-3hz  1898015274778547
  0  41241                     :profile-3hz  1898015608118262
  0  41241                     :profile-3hz  1898015941430060
  1  41241                     :profile-3hz  1898015275499014
  1  41241                     :profile-3hz  1898015609173485
  1  41241                     :profile-3hz  1898015942505828
  2  41241                     :profile-3hz  1898015275351257
  2  41241                     :profile-3hz  1898015609180861
  2  41241                     :profile-3hz  1898015942512708
  3  41241                     :profile-3hz  1898015274803528
  3  41241                     :profile-3hz  1898015608120522
  3  41241                     :profile-3hz  1898015941449884
^C
```

The time stamps printed are not in the correct order. Time stamp jumps occur when DTrace collects data from a different CPU buffer, visible as a change in CPU ID in the first column.

Tip 6: Use Perl to Postprocess

DTrace is capable of processing and presenting data in powerful ways, as demonstrated by many of the scripts in this book. This includes associating thread events to calculate delta times, using aggregations to print distribution plots, speculative tracing to include output based on some later event, and much more. You could spend a lot of time trying these features to achieve the desired output, but in some cases it may be quicker to dump raw data and postprocess it in another language, such as Perl.

We're all for using DTrace as much as possible, and this can result in some impressive standalone scripts. The point of this tip is to be practical: If you really need to solve a problem quickly, there are a few advantages to simplifying a DTrace script and post-processing its output.

- The DTrace part should become simple and quick to write, for example, `printf()` statements.
- The output can be reprocessed in different ways to produce different reports.
- You may already know languages such as Perl really well.

Here's an example of dumping potentially useful information from the io provider:

```
# dtrace -n 'io:::start,io:::done { printf("%d %d %s %s %x %d %d %d", timestamp, pid,
execname, args[1]->dev_statname, arg0, args[0]->b_bcount, args[0]->b_blkno, args[0]->
b_flags); }'
dtrace: description 'io:::start,io:::done ' matched 10 probes
CPU     ID                    FUNCTION:NAME
  0  24014    bdev_strategy:start 141488280383085 0 sched sd0 ffffff030a5c1e40 8192 16
95648 17301761
  0  24002           biodone:done 141488281001231 0 sched sd0 ffffff030a5c1e40 8192 16
95648 50856193
  0  24014    bdev_strategy:start 141488281130011 0 sched sd0 ffffff030a5c1e40 1536 17
04449 17301761
  0  24014    bdev_strategy:start 141488281226254 0 sched sd0 ffffff03371f7800 1536 56
10498 17301761
  0  24002           biodone:done 141488281292792 0 sched sd0 ffffff030a5c1e40 1536 17
04449 50856193
  0  24002           biodone:done 141488281575235 0 sched sd0 ffffff03371f7800 1536 56
10498 50856193
  0  24014    bdev_strategy:start 141488282042493 0 sched sd0 ffffff03371f7800 32256 1
704520 17301761
  0  24002           biodone:done 141488282253075 0 sched sd0 ffffff03371f7800 32256 1
704520 50856193
```

```
  0  24014    bdev_strategy:start 141488302929675 0 sched sd0 ffffff0304da7c80 131072
1650176 17301761
  0  24014    bdev_strategy:start 141488303307458 0 sched sd0 ffffff0336e4c580 131072
1650432 17301761
...
```

Imagine you're debugging an intermittent issue that occurs only once a day. While developing DTrace scripts, it may take several iterations to get it to output the desired summary. If you can test a script only once a day, it could take several days to develop the desired DTrace script. Instead, you could write a simple DTrace script that dumps everything of possible interest using `printf()` statements, across whichever probes seem interesting. Then spend time developing a Perl program to postprocess, without waiting for the next intermittent occurrence of the issue. If the intermittent problem vanishes, you still have the raw data to continue your analysis.

Tip 7: Learn Syscalls

By tracing system calls, you can examine all application I/O as well as file system and process operations. Because system calls have a reasonably stable interface that is also well documented (man pages), they provide excellent probe points for use with DTrace. By learning system calls well, you may also discover some clever uses of syscall provider probes.

For example, the syscall provider can be used to discover application configuration files by tracing all `open()` syscalls while the application is launched. Here `sshd` (the SSH daemon) is examined while it is restarted:

```
# dtrace -n 'syscall::open*:entry /execname == "sshd"/ { @[copyinstr(arg0)] =
count(); }'
dtrace: description 'syscall::open*:entry ' matched 2 probes
^C

  /dev/conslog                                                        1
  /dev/null                                                           1
  /dev/tty                                                            1
  /etc/default/login                                                  1
  /etc/netconfig                                                      1
  /etc/ssh/ssh_host_dsa_key                                           1
  /etc/ssh/ssh_host_rsa_key                                           1
  /etc/ssh/sshd_config                                                1
  /lib/libbsm.so.1                                                    1
  /lib/libcrypto.so.0.9.8                                             1
  /lib/libgen.so.1                                                    1
  /lib/libnsl.so.1                                                    1
  /lib/libsocket.so.1                                                 1
  /lib/svc/method/sshd                                                1
  /usr/lib/libgss.so.1                                                1
  /usr/share/lib/zoneinfo/UTC                                         1
```

```
    /var/run/sshd.pid                                                1
    /dev/udp                                                         2
    /lib/libc.so.1                                                   2
    /var/ld/ld.config                                                2
    /var/run/syslog_door                                             2
```

The configuration files under /etc have been identified, along with other files of interest, including the PID file in /var/run.

A trickier example was shown in Chapter 7, Network Protocols, for tracing SSH logins via the chdir() syscall:

```
server# dtrace -n 'syscall::chdir:entry /execname == "sshd"/ { printf("UID:%d %s",
uid, copyinstr(arg0)); }'
dtrace: description 'syscall::chdir:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  9  14265                      chdir:entry UID:130948 /home/brendan
```

This assumes that sshd executes chdir() to the user home directory after becoming the user.

Other clever uses have been shown in this book, especially for cases where a stable DTrace provider is not available and the syscall provider is the next best option.

Tip 8: timestamp vs. vtimestamp

DTrace provides two nanosecond timestamp variables, timestamp and vtimestamp. timestamp is elapsed time since system boot, in nanoseconds. vtimestamp is also nanoseconds but begins at thread creation and is incremented only when that thread is on-CPU.

The delta between two measurements of these timestamp types can answer the following:

- timestamp2 – timestamp1: Elapsed time, wall clock time, or latency
- vtimestamp2 – vtimestamp1: On-CPU time

Knowing these deltas can lead to areas of further analysis for understanding latency.

As an example, the following code calculates these deltas for the read() syscall. (A system call is chosen for this example as it makes association between the points easy: self-> variables can be used without worrying about recursive entry.)

```
syscall::read:entry
{
      self->start = timestamp;
      self->vstart = vtimestamp;
}

syscall::read:return
/self->start/
{
      this->elapsed = timestamp - self->start;
      this->oncpu = vtimestamp - self->vstart;
...
```

Now that elapsed and on-CPU time are known, the following interpretation can be applied.

`elapsed` ~= `oncpu`: Latency is due to on-CPU time.

`elapsed` >> `oncpu`: Latency is due to off-CPU time.

This determination points to areas we can analyze to understand latency further:

On-CPU time: Hot code paths, lock contention, CPU cross calls, memory bus I/O, and so on

Off-CPU time: Disk I/O, network I/O, lock wait, CPU dispatcher queue latency, and so on

So, the `timestamp` and `vtimestamp` deltas are very useful to know and compare.

Tip 9: profile:::profile-997 and Profiling

The `profile` probe (from the profile provider) allows DTrace to sample on all CPUs at a custom interval, specified in Hertz if no units are given. It's best to use an odd- or unusually numbered interval (`profile-997`, `profile-1001`, `profile-1234`), instead of a round number such as `profile-1000` (sample every millisecond), to avoid sampling in lockstep with any scheduled task that is also running every millisecond, which would unfairly inflate (or deflate) any activity measured in the samples.

Profiling software in this way is a quick and effective technique to see where CPU cycles are spent. The action taken when the profile probe fires can be to sample the function or stack trace that is on-CPU. Profiling was used in Chapter 9, Applications, to profile user-level software and in Chapter 12, Kernel, to profile the kernel. Both of these were performed with one-liners, which are among the most useful (and frequently used) DTrace one-liners:

User stack trace profile at 101 Hertz, showing process name and stack:

```
dtrace -n 'profile-101 { @[execname, ustack()] = count(); }'
```

Kernel stack trace profile at 1001 Hertz:

```
dtrace -n 'profile-1001 { @[stack()] = count(); }'
```

Tip 10: Variable Scope and Use

Recall DTrace provides different types of user-defined variables that differ in scope:

- Global variables, such as `variable_name`
- Thread-local variables, such as `self->variable_name`
- Clause-local variables, such as `this->variable_name`
- Aggregation variables, such as `@variable_name`

Thread-local variables (`self->`) can be referenced by different probes in the same thread context, whereas clause-local variables (`this->`) can be referenced only in action blocks for the same probe. The performance impact of clause-local variables is lower, so they should be used whenever possible, such as for temporary calculations in an action block.

Some DTrace scripts use thread-local variables to contain temporary strings in an action block, only because the first release of Solaris 10 did not allow clause-local string variables.

Thread-Local Variables

Thread-local variables can be referenced by different probes in the same thread context. This can be very useful when coordinating different events such as a system call being made by an application and the kernel functions involved in executing that system call. There are certain instances, however, in which seemingly related probes don't necessarily fire in the same thread context.

The io provider `start` and `done` probes are a good example of probes that generally fire in the same thread. It might be tempting to use the following script to gather statistics on how long individual I/Os are taking to complete:

```
#!/usr/sbin/dtrace -qs

io:::start
{
    self->ts = timestamp;
}

io:::done
/self->ts/
{
    @ = avg(timestamp - self->ts);
    self->ts = 0;
}
```

Unfortunately, this script will give the wrong data. Although it's possible that `io:::start` might fire synchronously with respect to the thread causing this I/O, the `io:::done` probe will not. When the I/O completes, the `done` probe will fire in the context of whichever thread happens to be on-CPU at the time. Because the thread that initiated this I/O is sleeping waiting for the I/O to finish, it will never be the case that the `done` probe fires in the same thread in which the `start` probe fired.

When using probes that fire in different thread contexts, you need to find some unique identifier associated with this probe and use a global array indexed on that identifier. For the io provider, the device and block number are useful as a unique identifier. The previous script would be rewritten as follows:

```
#!/usr/sbin/dtrace -ws

io:::start
{
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    @ = avg(timestamp - start[args[0]->b_edev, args[0]->b_blkno]);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}
```

Instead of using a thread-local variable, we implemented a global variable (`start`) using data made available from the probe arguments that will be unique for a given firing of the `io:::start` probe.

Clause-Local Variables

Although intended for use within a single clause, since clause-local variables (`this->`) are not freed at the end of a clause, they may be accessed in other clauses with the same probe name.

Global and Aggregation Variables

Although global variables may work, try to use aggregate variables instead. For example, any time a counter is needed such as x++, use an aggregation instead: @x = count(). This will usually be possible if the counter is gathering the data to be printed; it may not be possible if the counter must be used within a predicate or an arithmetic expression.

The reason for this is that aggregations are designed to be multi-CPU safe, whereas the global variables are not. There are cases where a global variable can become invalid when written to by DTrace actions firing on multiple CPUs.

As an example, consider the following DTrace script:

```
#!/usr/sbin/dtrace -qs

profile-997
{
    total++;
    @ = count();
}

END
{
    printf("Global == %d\n", total);
    printf("Aggregate == %@d\n", @);
}
```

On a multi-CPU system, we would expect to see the global variable and the value stored in the aggregation deviate because of the method involved in updating the global variable. Even on a system with only two CPUs, we can hit this situation very frequently, as shown in the following output:

```
# ./global.d
^C
Global == 10661
Aggregate == 15254

#
```

Tip 11: strlen() and strcmp()

These functions and the strings that they process can be traced via the pid provider, which can sometimes help navigate an unfamiliar body of code.

For example, the argument to strlen() and the user stack trace were traced for a bash shell, while ls -l was typed in that shell. One of the stacks discovered was the following:

```
# dtrace -n 'pid$target::strlen:entry { @[copyinstr(arg0), ustack()] =
count(); }' -p 592
dtrace: description 'pid$target::strlen:entry ' matched 2 probes
^C
[... output truncated ...]
  ls -l
              libc.so.1`strlen
              bash`alloc_history_entry+0x23
              bash`add_history+0xdd
              bash`really_add_history+0x22
              bash`bash_add_history+0x114
              bash`check_add_history+0x52
              bash`maybe_add_history+0x57
              bash`pre_process_line+0xe6
              bash`shell_getc+0x312
              bash`read_token+0x3f
              bash`yylex+0x95
              bash`yyparse+0x2c1
              bash`parse_command+0x64
              bash`read_command+0xb2
              bash`reader_loop+0x11b
              bash`main+0x6dd
              bash`_start+0x7d
                1
```

The output includes the `ls -l` command string, implying that this stack trace is related to the processing of commands. Reading the stack trace from the bottom up shows this `strlen()` was used while adding the command to the `bash` history.[2] So, by simply tracing `strlen()`, we now have an idea of the code flow within `bash` for this particular action (adding history). `strcmp()` and the other string functions may also be used for this type of experiment.

Tip 12: Check Assumptions

DTrace lets you check your assumptions, usually with short one-liners. Try to get into the habit of not only being aware of the assumptions you are making but also checking them where possible with DTrace.

For example, you might assume that operating system statistics such as network interface statistics are always correct. This isn't true: Bugs happen. DTrace can be used to calculate statistics from different points in the system to double-check their accuracy. (This has unearthed statistics bugs on more than one occasion.)

2. Yes, this stack trace is real and is one of my favorites.

Tip 13: Keep It Simple

And stable. It's possible to write long and complex DTrace scripts, especially when navigating the thousands of available probes from the fbt and pid providers. An example of this is the fbt-based `tcpsnoop.d`, shown and explained in Chapter 6, Network Lower-Level Protocols. Although you can write scripts like this, try to solve your tracing needs with short and simple scripts instead.

Long scripts become difficult to maintain. And if they use unstable providers such as fbt and pid, they will need maintenance to match changes in the target software. Always check for the availability of stable providers first, because more are being written each year.

It's also easy to write incorrect DTrace scripts. With DTrace, you can quickly go from having no visibility in an area to producing numerous custom statistics. Without careful checking and testing, these statistics can be dead wrong, and if you previously had no visibility into an area, it may not be obvious that the statistics are wrong. Short and simple scripts are easier to check and verify.

Tip 14: Consider Performance Impact

DTrace has been designed to minimize its impact on performance. This design includes the following:

- Per-CPU kernel buffers that are read by user-land `dtrace` at a slow rate (switchrate)
- Dropping events when the rate is too high
- "Abort due to systemic unresponsiveness"

On systems with a large number of CPUs, the startup cost of DTrace can be minimized by tuning down the principal buffer size (4MB by default). This may require some incremental steps to avoid DTrace warning of data drops (see tip 15), but it is very easy to do either on the command line or within a script:

```
# dtrace -b 512k -n 'syscall:::entry { @[execname,probefunc] = count(); }'
```

or

```
# dtrace -x bufsize=512k -n 'syscall:::entry { @[execname,probefunc] = count(); }'
```

or, in a D script:

```
#pragma D option bufsize=512k
```

However, you should keep in mind that there may still be a small impact on performance. Enabled DTrace probes have a small CPU cost for execution, which can affect performance when DTrace probes are firing frequently. The easiest way to determine this impact is to run `dtrace` for a number of seconds as an experiment and to measure the performance loss while `dtrace` is running. To ensure `dtrace` stops running after a number of seconds, the profile provider `tick` probe can be used to call `exit()` after the desired interval.

Taking that one step further, it can be useful when using DTrace on a busy production system to leverage the ability to specify very short time durations with the tick probe, to ensure DTrace will exit quickly, and to gradually increase the duration once it is determined that the D being executed is not inducing application issues. For example, start with this:

```
tick-1sec
{
      exit(0);
}
```

in your DTrace script (or even `tick-500msec`). If the D is running as expected, increase the duration to capture a more meaningful sample.

In general, be careful when probes are firing more than 10,000 times per second. The impact also depends on the CPU horsepower of the target system and the complexity of the DTrace actions. Actions that have a higher performance cost include `copyin()` and `copyinstr()`, which copy data from the user to the kernel address space.

The pid provider can especially hurt performance if misused, because it can trace not only every function entry and return in user-level software but also every instruction—potentially millions of probes. Caution should be taken to account for the number of probes enabled and their frequency. (See Chapter 9, Applications, for more discussion on the pid provider.)

Performance can also suffer when outputting large volumes of trace data to an X Windows screen on the same server that is being DTraced (where performance impact is due to screen updates).

Although there are scenarios where the performance cost may be high (for example, tracing details of all `malloc()` calls by a busy application), the information retrieved by DTrace may nonetheless be worth the cost.

Tip 15: drops and dynvardrops

If the DTrace main buffer overflows because of the system's inability to drain it quickly enough, "drops" warnings will be printed (for example, `dtrace: 710 drops on CPU 0`). To eliminate these warnings, the principal buffer size can be increased with either the `-b` option at the command line or the `bufsize` tunable option. The default is 4MB per CPU. Another fix may be to increase the `switchrate` tunable option to flush the buffers more quickly than the default of once per second.

Another type of warning is `dynvardrops`, when the dynamic variable buffer overflows. The `dynvarsize` tunable can be used to increase the size of this buffer. This often happens when assigning dynamic variables and then forgetting to free them after they are no longer needed, and over time they can fill the dynamic variable buffer to the point of `dynvardrops`. Rather than increase the `dynvarsize` variable, first inspect your D script to determine whether there are variables you should be freeing but are not. Note `dynvardrops` must be eliminated for correct results.

Tip 16: Tail-Call Optimization

This is a compiler optimization feature to reuse the caller's stack frame when one function ends by calling another function. Although this saves register window instructions, it causes a problem for DTrace—the function return probe will fire before the entry probe. This happens more frequently on SPARC than x86 platforms.

There is another optimization you may encounter that has the side effect of a function return probe not firing at all. This can happen when a function returns into its parent function, which then immediately returns; the compiler can optimize the first return to skip past the second to save instructions. As a consequence, DTrace sees the function entry probe fire but not the return probe.

Further Reading

See *Advanced DTrace Tips, Tricks and Gotchas* by Bryan Cantrill, Mike Shapiro, and Adam Leventhal.[3]

3. *http://dtrace.org/blogs/bmc/2005/02/28/dtrace-tips-tricks-and-gotchas/*

This page intentionally left blank

A

DTrace Tunable Variables

Several tunable variables are available within DTrace for customization when necessary. Many of these variables are named at the DTrace consumer level to facilitate per-consumer modifications (that is, per instance of `dtrace(1M)` either as a one-liner or as a D script), with a corresponding kernel variable name that will have systemwide scope (affecting every instance of `dtrace(1M)`). The default values work very well the majority of the time. For the most part, changing the default value can and should be done on a per-consumer basis vs. systemwide. Table A-1 is taken from the "Options and Tunables" chapter of the DTrace Guide,[1] with some additional information, such as the default values on Solaris 10.

The different ways to set tunable variables are as follows:

Per-consumer, command line: `-x` ***`consumer_variable_name=value`***

Per-consumer, D script: `#pragma D option` ***`consumer_variable_name=value`***

Systemwide, Solaris `/etc/system`: `set` ***`kernel_variable_name`***`=`***`value`***

As listed in Table A-1, some tunable variables have command-line aliases for convenience, for example, using `-b` ***`size`*** instead of `-x bufsize=`***`size`***. Note that

1. See the bibliography for the current location of the DTrace Guide.

a few of the tunables are simply set and do not require a value to be specified; these are those with "Disabled" in the Default Value column, for example, using either `-q` or `-x quiet` for quiet mode.

Table A-1 DTrace Tunable Variables

Consumer Variable Name	Kernel Tunable Variable Name	Default Value	`dtrace(1M)` Alias	Description
`aggrate`	`dtrace_aggrate_default`	1Hz	None	Rate at which aggregation buffers are read.
`Aggsize`	None	4MB	None	The per-CPU size of aggregation buffers.
`bufresize`	None	Auto	None	Buffer resizing policy. Optional setting of `manual` will cause DTrace to fail to start if an allocation failure occurs. This variable affects all DTrace buffers. See the "Buffers and Buffering" chapter in the DTrace Guide.
`bufsize`	None	4MB	`-b`	The per-CPU size of principal buffers.
`bufpolicy`	None	Switch	None	The buffer management policy used. Optional settings are `fill` and `ring`. See the "Buffers and Buffering" chapter in the DTrace Guide.
`cleanrate`	`dtrace_cleanrate_default`	101Hz	None	The rate at which speculative buffers are cleaned. See the "Speculations" section in Chapter 2.
`cpu`	None	None—all CPUs	`-c`	The CPU on which to enable tracing. By default, buffer allocation and tracing is enabled for all CPUs.

continues

Table A-1 DTrace Tunable Variables (*Continued*)

Consumer Variable Name	Kernel Tunable Variable Name	Default Value	dtrace(1M) Alias	Description
defaultargs	None	Disabled	None	Allow use of $1..$N macro variables while they are undefined at the command line; for which integers default to zero, strings to NULL.
destructive	dtrace_destructive_disallow	Disabled	-w	Allow destructive actions, including raise() and panic(). See Appendix B.
dynvarsize	dtrace_dstate_defsize	1MB	None	Dynamic variable space size.
flowindent	None	Disabled	-F	Indent function entry and prefix with ->; unindent function return and prefix with <-.
grabanon	None	Disabled	-a	Claim anonymous state. See the "Anonymous Tracing" section in Chapter 12.
jstackframes	dtrace_jstackframes_default	50	None	Maximum number of default jstack() stack frames.
jstackstrsize	dtrace_jstackstrsize_default	512 bytes	None	Default string space size for jstack().
nspec	dtrace_nspec_default	1	None	Number of speculations (number of speculative buffers).
quiet	None	Disabled	-q	When enabled, output only explicitly traced data.
rawbytes	None	Disabled	None	When enabled, tracemem generates only hexidecimal data.
specsize	dtrace_specsize_default	32KB	None	Size of speculation buffers.

continues

Table A-1 DTrace Tunable Variables (*Continued*)

Consumer Variable Name	Kernel Tunable Variable Name	Default Value	`dtrace(1M)` Alias	Description
`strsize`	`dtrace_strsize_default`	256 bytes	None	Size available for string variables.
`stackframes`	`dtrace_stackframes_default`	20	None	Maximum number of kernel stack frames (`stack()`).
`stackindent`	None	14	None	Number of whitespace characters to use when indenting `stack()` and `ustack()` output.
`statusrate`	`dtrace_statusrate_default`	1Hz	None	Rate of status checking.
`switchrate`	`dtrace_switchrate_default`	1Hz	None	Rate of switch buffer switching.
`ustackframes`	`dtrace_ustackframes_default`	20	None	Maximum number of user stack frames (`ustack()`).

The `dumpvars.d` script (shown here) can be executed on your target system to dump the current values of DTrace kernel tunable variables. Note this script does not work on FreeBSD 8.0.

```
solaris# ./dumpvars.d
dtrace_destructive_disallow:  0
dtrace_nonroot_maxsize:       16777216
dtrace_difo_maxsize:          262144
dtrace_dof_maxsize:           262144
dtrace_global_maxsize:        16384
dtrace_actions_max:           16384
dtrace_retain_max:            1024
dtrace_helper_actions_max:    32
dtrace_helper_providers_max:  32
dtrace_dstate_defsize:        1048576
dtrace_strsize_default:       256
dtrace_cleanrate_default:     9900990
dtrace_cleanrate_min:         200000
dtrace_cleanrate_max:         60000000000
dtrace_aggrate_default:       1000000000
dtrace_statusrate_default:    1000000000
dtrace_statusrate_max:        10000000000
dtrace_switchrate_default:    1000000000
```

```
dtrace_nspec_default:         1
dtrace_specsize_default:      32768
dtrace_stackframes_default:   20
dtrace_ustackframes_default:  20
dtrace_jstackframes_default:  50
dtrace_jstackstrsize_default: 512
dtrace_msgdsize_max:          128
dtrace_chill_max:             500000000
dtrace_chill_interval:        1000000000
dtrace_devdepth_max:          32
dtrace_err_verbose:           0
dtrace_deadman_interval:      1000000000
dtrace_deadman_timeout:       10000000000
dtrace_deadman_user:          30000000000
```

```
macosx# ./dumpvars.d
dtrace_destructive_disallow:  0
dtrace_nonroot_maxsize:       16777216
dtrace_difo_maxsize:          262144
dtrace_dof_maxsize:           393216
dtrace_global_maxsize:        16384
dtrace_actions_max:           16384
dtrace_retain_max:            1024
dtrace_helper_actions_max:    32
dtrace_helper_providers_max:  32
dtrace_dstate_defsize:        1048576
dtrace_strsize_default:       256
dtrace_cleanrate_default:     9900990
dtrace_cleanrate_min:         200000
dtrace_cleanrate_max:         60000000000
dtrace_aggrate_default:       1000000000
dtrace_statusrate_default:    1000000000
dtrace_statusrate_max:        10000000000
dtrace_switchrate_default:    1000000000
dtrace_nspec_default:         1
dtrace_specsize_default:      32768
dtrace_stackframes_default:   20
dtrace_ustackframes_default:  20
dtrace_jstackframes_default:  50
dtrace_jstackstrsize_default: 512
dtrace_msgdsize_max:          128
dtrace_chill_max:             500000000
dtrace_chill_interval:        1000000000
dtrace_devdepth_max:          32
dtrace_err_verbose:           0
dtrace_deadman_interval:      1000000000
dtrace_deadman_timeout:       10000000000
dtrace_deadman_user:          30000000000
```

```
#!/usr/sbin/dtrace -qs
dtrace:::BEGIN
{
      printf("dtrace_destructive_disallow:  %d\n",`dtrace_destructive_disallow);
      printf("dtrace_nonroot_maxsize:       %d\n",`dtrace_nonroot_maxsize);
      printf("dtrace_difo_maxsize:          %d\n",`dtrace_difo_maxsize);
      printf("dtrace_dof_maxsize:           %d\n",`dtrace_dof_maxsize);
      printf("dtrace_global_maxsize:        %d\n",`dtrace_global_maxsize);
      printf("dtrace_actions_max:           %d\n",`dtrace_actions_max);
      printf("dtrace_retain_max:            %d\n",`dtrace_retain_max);
```

continues

```
        printf("dtrace_helper_actions_max:    %d\n",`dtrace_helper_actions_max);
        printf("dtrace_helper_providers_max:  %d\n",`dtrace_helper_providers_max);
        printf("dtrace_dstate_defsize:        %d\n",`dtrace_dstate_defsize);
        printf("dtrace_strsize_default:       %d\n",`dtrace_strsize_default);
        printf("dtrace_cleanrate_default:     %d\n",`dtrace_cleanrate_default);
        printf("dtrace_cleanrate_min:         %d\n",`dtrace_cleanrate_min);
        printf("dtrace_cleanrate_max:         %d\n",`dtrace_cleanrate_max);
        printf("dtrace_aggrate_default:       %d\n",`dtrace_aggrate_default);
        printf("dtrace_statusrate_default:    %d\n",`dtrace_statusrate_default);
        printf("dtrace_statusrate_max:        %d\n",`dtrace_statusrate_max);
        printf("dtrace_switchrate_default:    %d\n",`dtrace_switchrate_default);
        printf("dtrace_nspec_default:         %d\n",`dtrace_nspec_default);
        printf("dtrace_specsize_default:      %d\n",`dtrace_specsize_default);
        printf("dtrace_stackframes_default:   %d\n",`dtrace_stackframes_default);
        printf("dtrace_ustackframes_default:  %d\n",`dtrace_ustackframes_default);
        printf("dtrace_jstackframes_default:  %d\n",`dtrace_jstackframes_default);
        printf("dtrace_jstackstrsize_default: %d\n",`dtrace_jstackstrsize_default);
        printf("dtrace_msgdsize_max:          %d\n",`dtrace_msgdsize_max);
        printf("dtrace_chill_max:             %d\n",`dtrace_chill_max);
        printf("dtrace_chill_interval:        %d\n",`dtrace_chill_interval);
        printf("dtrace_devdepth_max:          %d\n",`dtrace_devdepth_max);
        printf("dtrace_err_verbose:           %d\n",`dtrace_err_verbose);
        printf("dtrace_deadman_interval:      %d\n",`dtrace_deadman_interval);
        printf("dtrace_deadman_timeout:       %d\n",`dtrace_deadman_timeout);
        printf("dtrace_deadman_user:          %d\n",`dtrace_deadman_user);

        exit(0);
}
```

Script dumpvars.d

B

D Language Reference

To provide the most complete reference possible, Tables B-1 through B-18, listing built-in variables and built-in functions, are based on Solaris Nevada, circa June 2010, which has the most available. It is possible that some of the variables and/or functions listed in these tables will not be available, depending on which operating system, and which version of a specific operating system, is being used.

Table B-1 Built-in Variables

Type and Name	Description
`int64_t arg0, ..., arg9`	The first ten input arguments to a probe represented as raw 64-bit integers. If fewer than ten arguments are passed to the current probe, the remaining variables return zero.
`args[]`	The typed arguments to the current probe, if any. The `args[]` array is accessed using an integer index, but each element is defined to be the type corresponding to the given probe argument (if type information is available). For example, if `args[]` is referenced by a `read(2)` system call probe, `args[0]` is of type `int`, `args`[1] is of type `void *`, and `args[2]` is of type `size_t`.
`uintptr_t caller`	The program counter location of the current kernel thread at the time the probe fired.

continues

Table B-1 Built-in Variables (*Continued*)

Type and Name	Description
`uintptr_t ucaller`	The program counter location of the current user thread at the time the probe fired.
`chipid_t chip`	The CPU chip identifier for the current physical chip.
`processorid_t cpu`	The CPU identifier for the current CPU.
`cpuinfo_t *curcpu`	The CPU information for the current CPU.
`lwpsinfo_t *curlwpsinfo`	The lightweight process (LWP) state of the LWP associated with the current thread.
`psinfo_t *curpsinfo`	The process state of the process associated with the current thread.
`kthread_t *curthread`	The address of the operating system kernel's internal data structure for the current thread; for Solaris, the `kthread_t`. The `kthread_t` is defined in `<sys/thread.h>`. Refer to *Solaris Internals* for more information on this variable and other operating system data structures.
`string cwd`	The name of the current working directory of the process associated with the current thread.
`uint_t epid`	The enabled probe ID (EPID) for the current probe. This integer uniquely identifies a particular probe that is enabled with a specific predicate and set of actions.
`int errno`	The error value returned by the last system call executed by this thread.
`string execname`	The name that was passed to `exec(2)` to execute the current process.
`gid_t gid`	The real group ID of the current process.
`uint_t id`	The probe ID for the current probe. This ID is the system-wide unique identifier for the probe as published by DTrace and listed in the output of `dtrace -l`.
`uint_t ipl`	The interrupt priority level (IPL) on the current CPU at probe firing time. Refer to *Solaris Internals* for more information on interrupt levels and interrupt handling in the Solaris operating system kernel.
`lgrp_id_t lgrp`	The latency group ID for the latency group of which the current CPU is a member.
`pid_t pid`	The process ID of the current process.
`pid_t ppid`	The parent process ID of the current process.

Table B-1 Built-in Variables (*Continued*)

Type and Name	Description
`string probefunc`	The function name portion of the current probe's description.
`string probemod`	The module name portion of the current probe's description.
`string probename`	The name portion of the current probe's description.
`string probeprov`	The provider name portion of the current probe's description.
`psetid_t pset`	The processor set ID for the processor set containing the current CPU.
`string root`	The name of the root directory of the process associated with the current thread.
`uint_t stackdepth`	The current thread's kernel stack frame depth at probe firing time.
`id_t tid`	The thread ID of the current thread. For threads associated with user processes, this value is equal to the result of a call to `pthread_self(3C)`.
`uint64_t timestamp`	The current value of a nanosecond timestamp counter. This counter increments from an arbitrary point in the past and should be used only for relative computations.
`uintptr_t ucaller`	The program counter location of the current user thread at the time the probe fired.
`uid_t uid`	The real user ID of the current process.
`uint64_t uregs[]`	The current thread's saved user-mode register values at probe firing time. Use of the `uregs[]` array is discussed in the "User Process Tracing" chapter of the DTrace Guide.
`uint64_t ustackdepth`	The current thread's user stack depth.
`uint64_t vtimestamp`	The current value of a nanosecond timestamp counter that is virtualized to the amount of time that the current thread has been running on a CPU, minus the time spent in DTrace predicates and actions. This counter increments from an arbitrary point in the past and should be used only for relative time computations.
`uint64_t walltimestamp`	The current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.
`string zonename`	The name of the zone.

Table B-2 Built-in Functions

Function Name and Prototype	Description
Subroutines	
`void *alloca(size_t size)`	Allocates size bytes out of scratch space. Returns a pointer to the allocated space.
`string basename(char *str)`	Creates a copy of the string `str`, without a prefix that ends in /.
`void bcopy(void *src, void *dest, size_t size)`	Copies size bytes from `src` address to `dest` address.
`string cleanpath(char *str)`	Creates a string that consists of a copy of the path pointed to by `str`, but with certain redundant elements removed and with /./ and /../ elements collapsed.
`void *copyin(uintptr_t addr, size_t size)`	Copies the specified size in bytes from the specified user address into a DTrace scratch buffer and returns the address of this buffer
`string copyinstr(uintptr_t addr)` `string copyinstr(uintptr_t addr, size_t maxlength)`	Copies a null-terminated C string from the specified user address into a DTrace scratch buffer and returns the address of this buffer.
`void copyinto(uintptr_t addr, size_t size, void *dest)`	Copies the specified size in bytes from the specified user address into the DTrace scratch buffer specified by `dest`.
`string ddi_pathname(dev_info_t *, minor_number)`	Returns the device pathname for the `dev_info_t` and device minor number.
`string dirname(char *str)`	Creates a string that consists of all but the last level of the pathname specified by `str`.
`void exit(int status)`	The exit action is used to immediately stop tracing and exit the consumer.
`void ftruncate()`	Truncates STDOUT.
`_symaddr func(uintptr_t addr)`	Returns the kernel function associated with `addr`.
`int getmajor(dev_t)`	Returns the major number of the device referenced by `dev_t`.
`int getminor(dev_t)`	Returns the minor number of the device referenced by `dev_t`
`uint32_t htonl(uint32_t)`	Converts a 32-bit value from host byte order to network byte order.
`uint64_t htonll(uint64_t)`	Converts a 64-bit value from host byte order to network byte order.

Table B-2 Built-in Functions (*Continued*)

Function Name and Prototype	Description
`uint16_t htons(uint16_t)`	Converts a 16-bit value from host byte order to network byte order.
`int index(string, char)`	Returns a pointer to the first occurrence of char in string.
`string inet_ntoa(ipaddr_t *addr)`	Takes a pointer to an IPv4 address and returns it as a dotted quad decimal string.
`string inet_ntoa6(in6_addr_t *addr)`	Takes a pointer to an IPv6 address and returns it as an RFC 1884 convention 2 string, with lowercase hexadecimal digits.
`string inet_ntop(int af, void *addr)`	Takes a pointer to an IP address and returns a string version depending on the provided address family.
`string lltostr(uint64_t)`	Returns a pointer to a string represented by 64-bit unsigned integer value.
`_symaddr mod(address)`	Returns the kernel module associated with address.
`size_t msgdsize(mblk_t *mp)`	Returns the number of bytes in the data message pointed to by `mp`.
`size_t msgsize(mblk_t *mp)`	Returns the number of bytes in the message pointed to by `mp`: total bytes, not just data bytes.
`int mutex_owned(kmutex_t *mutex)`	Returns nonzero if the calling thread currently holds the specified kernel mutex, or returns zero if the specified adaptive mutex is currently unowned.
`kthread_t *mutex_owner(kmutex_t *mutex)`	Returns the thread pointer of the current owner of the specified adaptive kernel mutex. `mutex_owner` returns `NULL` if the specified adaptive mutex is currently unowned or if the specified mutex is a spin mutex.
`int mutex_type_adaptive(kmutex_t *mutex)`	Returns nonzero if the specified kernel mutex is of type `MUTEX_ADAPTIVE` or zero if it is not.
`int mutex_type_spin(kmutex_t *)`	Returns nonzero if the specified kernel mutex is of type `MUTEX_SPIN` or zero if it is not.
`uint32_t ntohl(uint32_t)`	Converts a 32-bit value from network byte order to host byte order.
`uint64_t ntohll(uint64_t)`	Converts a 64-bit value from network byte order to host byte order.
`uint16_t ntohs(uint16_t)`	Converts a 16-bit value from network byte order to host byte order.

continues

Table B-2 Built-in Functions (*Continued*)

Function Name and Prototype	Description
`int progenyof(pid_t pid)`	Returns nonzero if the calling process (the process associated with the thread that is currently triggering the matched probe) is among the progeny of the specified process ID.
`int rindex(string, char)`	Returns a pointer to the last occurrence of char in string.
`int rand(void)`	Returns a pseudo-random integer.
`int rw_iswriter(krwlock_t *rwlock)`	Returns nonzero if the specified reader-writer lock is either held or desired by a writer. If the lock is held only by readers and no writer is blocked or if the lock is not held at all, `rw_iswriter` returns zero.
`int rw_read_held(krwlock_t *rwlock)`	Returns nonzero if the specified reader-writer lock is currently held by one or more readers, or zero otherwise.
`int rw_write_held(krwlock_t *rwlock)`	Returns nonzero if the specified reader-writer lock is currently held by a writer. If the lock is held only by readers or not held at all, `rw_write_held` returns zero.
`string strchr(string, char)`	Returns the first occurrence of char in string.
`string strjoin(string1, string2)`	Creates a string that consists of `str1` concatenated with `str2`.
`string strrchr(string, char)`	Returns the last occurrence of char in string.
`string strstr(string1, string2)`	Returns the position of the first occurrence of `string2` in `string1`.
`string strtok(string1, string2)`	Returns a token for string.
`size_t strlen(string)`	Returns the length of the specified string in bytes, excluding the terminating null byte.
`string substr(string, pos, len)`	Returns the substring of string starting at position `pos` for length `len`.
Data Recording Actions	
`_symaddr sym(uintptr_t address)`	Print the kernel symbol for the specified kernel address.
`void trace(expression)`	Takes a D expression as its argument and traces the result to the directed buffer

Table B-2 Built-in Functions (*Continued*)

Function Name and Prototype	Description
`void tracemem(address, size_t nbytes)`	Takes a D expression as its first argument, `address`, and a constant as its second argument, `nbytes`. `tracemem` copies the memory from the address specified by `addr` into the directed buffer for the length specified by `nbytes`. What happens when the buffer is processed depends on the size; 1, 2, 4, and 8 bytes will be printed as integers of that size; other sizes will be hex dumped.
`void printf(string format, ...)`	Print formatted.
`void printa(aggregation)` `void printa(string format, aggregation)`	Print an aggregation, with optional format specifiers.
`void stack(int nframes)` `void stack(void)`	The stack action records a *kernel* stack trace to the directed buffer.

`void ustack(int nframes, int strsize)`
`void ustack(int nframes)`
`void ustack(void)`
	The ustack action records a *user* stack trace to the directed buffer.

`void jstack(int nframes, int strsize)`
`void jstack(int nframes)`
`void jstack(void)`
	Java stack. `jstack` is an alias for `ustack` that performs in situ Java frame translation from the JVM. The `jstackframes` option tunes the number of stack frames, and the `jstackstrsize` option tunes the size of the string space used when generating `jstack`.

`_usymaddr uaddr(uintptr_t address)`	Prints the symbol for a specified user address, including hexadecimal offset.
`_usymaddr ufunc(uintptr_t address)`	Prints the user function for the specified user address.
`_usymaddr umod(uintptr_t address)`	Prints the user module for the specified user address.
`_usymaddr usym(uintptr_t address)`	Prints the user symbol for the specified user address.
Aggregation Functions (`@agg` denotes an aggregation variable)	
`@agg avg(int)`	Returns to an aggregation variable the arithmetic average

continues

Table B-2 Built-in Functions (*Continued*)

Function Name and Prototype	Description
`void clear(@agg)`	Clear an aggregation—clear all the values in the aggregation to zero. Does not remove the entries.
`@agg count()`	Returns to an aggregation variable the number of times called.
`void denormalize(@agg)`	Undo a prior `normalize()`.
`@agg lquantize(int, lower, upper, step)`	Returns to an aggregation variable a linear frequency distribution.
`@agg min(int)`	Returns to an aggregation variable the smallest value.
`@agg max(int)`	Returns to an aggregation variable the largest value.
`void normalize(@agg, int)`	Normalize the data in the aggregation by the passed normalization factor `int`.
`@agg quantize(int)`	Returns to an aggregation variable a power-of-two frequency distribution.
`void setopt(string option, char * pos)`	Sets aggregation sort option, with optional position `pos`. Aggregation sort options: `aggsortkey` sorts by key order. `aggsortrev` reverses sort order. `aggsortpos` sets the position of the aggregation to use as primary sort key (multiple aggregations). `aggsortkeypos` sets the position of key to use as primary sort key (multiple aggregations).
`@agg stddev(int)`	Returns to an aggregation variable the standard deviation.
`@agg sum(int)`	Returns to an aggregation variable the total value.
`void trunc(@agg)` `void trunc(@agg, int)`	Truncate an aggregation—remove all the aggregation entries (keys and values), or with optional `int`, truncate all but `int` entries.
Kernel Destructive Actions	
`void breakpoint(void)`	Induce a kernel breakpoint, transferring control to the kernel debugger.
`void chill(int nanoseconds)`	The chill action causes DTrace to spin for the specified number of nanoseconds.
`void panic(void)`	The panic action causes a kernel panic when triggered.

continues

Table B-2 Built-in Functions (*Continued*)

Function Name and Prototype	Description
Process Destructive Actions	
`void copyout(void *buf, uintptr_t addr, size_t nbytes)`	Copies `nbytes` from the buffer specified by `buf` to the address specified by `addr` in the address space of the process associated with the current thread.
`void copyoutstr(string str, uintptr_t addr, size_t maxlen)`	Copies the string specified by `str` to the address specified by `addr` in the address space of the process associated with the current thread.
`void freopen(string *)`	Redirects all writes to STDOUT to the specified file string *.
`void raise(int signal)`	The raise action sends the specified signal to the currently running process.
`void stop(void)`	The stop action forces the process that fires the enabled probe to stop when it next leaves the kernel.
`void system(string program, ...)`	Causes the program specified by `program` to be executed as if it were given to the shell as input.
Speculation Actions	
`id speculation()`	Returns an identifier for a new speculative buffer.
`void speculate(id)`	Denotes that the remainder of the clause should be traced to the speculative buffer specified by `id`.
`void commit(id)`	Commits the speculative buffer associated with `id`.
`void discard(id)`	Discards the speculative buffer associated with `id`.

Table B-3 Keywords

`auto*`	`do*`	`if*`	`register*`	`string+`	`unsigned`
`break*`	`double`	`import*+`	`restrict*`	`stringof+`	`void`
`case*`	`else*`	`inline`	`return*`	`struct`	`volatile`
`char`	`enum`	`int`	`self+`	`switch*`	`while*`
`const`	`extern`	`long`	`short`	`this+`	`xlate+`
`continue*`	`float`	`offsetof+`	`signed`	`translator+`	
`counter*+`	`for*`	`probe*+`	`sizeof`	`typedef`	
`default*`	`goto*`	`provider*+`	`static*`	`union`	

* Reserved for future use by the D language

\+ Defined by D but not defined by ANSI-C

Table B-4 Integer Data Types

Type Name	32-Bit Size	64-Bit Size
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes

Integer types may be prefixed with the `signed` or `unsigned` qualifier. If no sign qualifier is present, the type is assumed to be signed.

Table B-5 Integer Type Aliases

Type Name	Description
`int8_t`	1-byte signed integer
`int16_t`	2-byte signed integer
`int32_t`	4-byte signed integer
`int64_t`	8-byte signed integer
`intptr_t`	Signed integer of size equal to a pointer
`uint8_t`	1-byte unsigned integer
`uint16_t`	2-byte unsigned integer
`uint32_t`	4-byte unsigned integer
`uint64_t`	8-byte unsigned integer
`uintptr_t`	Unsigned integer of size equal to a pointer

Table B-6 Floating-Point Data Types

Type Name	32-Bit Size	64-Bit Size
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`long double`	16 bytes	16 bytes

Table B-7 Integer Suffixes

`u` or `U`	`unsigned` version of the type selected by the compiler
`l` or `L`	`long`
`ul` or `UL`	`unsigned long`
`ll` or `LL`	`long long`
`ull` or `ULL`	`unsigned long long`

Table B-8 Floating-Point Suffixes

`f` or `F`	`float`
`l` or `L`	`long double`

Table B-9 Character Escape Sequences

`\a`	Alert
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\?`	Question mark
`\'`	Single quote
`\"`	Double quote
`\0oo`	Octal value `oo`
`\xhh`	Hexadecimal value `hh`
`\0`	Null character

Table B-10 Binary Arithmetic Operators

`+`	Integer addition
`-`	Integer subtraction
`*`	Integer multiplication
`/`	Integer division
`%`	Integer modulus

Table B-11 D Unary Arithmetic Operators

++	Increment value
--	Decrement value

Table B-12 Binary Relational Operators

<	Left-hand operand is less than right-operand.
<=	Left-hand operand is less than or equal to right-hand operand.
>	Left-hand operand is greater than right-hand operand.
>=	Left-hand operand is greater than or equal to right-hand operand.
==	Left-hand operand is equal to right-hand operand.
!=	Left-hand operand is not equal to right-hand operand.

Table B-13 Binary Logical Operators

<table>
<tr><td>!</td><td>Logical negation of a single operand</td></tr>
<tr><td>&&</td><td>Logical AND: true if both operands are true</td></tr>
<tr><td>||</td><td>Logical OR: true if one or both operands are true</td></tr>
<tr><td>^^</td><td>Logical XOR: true if exactly one operand is true</td></tr>
</table>

Table B-14 Unary Logical Operators

!	Logical negation of a single operand

Table B-15 Binary Bitwise Operators

<table>
<tr><td>~</td><td>Bitwise negation of a single operand.</td></tr>
<tr><td>&</td><td>Bitwise AND.</td></tr>
<tr><td>|</td><td>Bitwise OR.</td></tr>
<tr><td>^</td><td>Bitwise XOR.</td></tr>
<tr><td><<</td><td>Shift the left-hand operand left by the number of bits specified by the right-hand operand.</td></tr>
<tr><td>>></td><td>Arithmetic-shift the left-hand operand right by the number of bits specified by the right-hand operand.</td></tr>
</table>

Table B-16 Unary Bitwise Operators

`~`	Bitwise negation of a single operand

Table B-17 Binary Assignment Operators

`=`	Set the left-hand operand equal to the right-hand expression value.
`+=`	Increment the left-hand operand by the right-hand expression value.
`-=`	Decrement the left-hand operand by the right-hand expression value.
`*=`	Multiply the left-hand operand by the right-hand expression value.
`/=`	Divide the left-hand operand by the right-hand expression value.
`%=`	Modulo the left-hand operand by the right-hand expression value.
`|=`	Bitwise OR the left-hand operand with the right-hand expression value.
`&=`	Bitwise AND the left-hand operand with the right-hand expression value.
`^=`	Bitwise XOR the left-hand operand with the right-hand expression value.
`<<=`	Shift the left-hand operand left by the number of bits specified by the right-hand expression value.
`>>=`	Shift the left-hand operand right by the number of bits specified by the right-hand expression value.

Table B-18 Operator Precedence and Associativity

Operators	Associativity
`() [] ->`	Left to right
`! ~ ++ − + - * & (type) sizeof stringof offsetof xlate`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`<< >>`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`^^`	Left to right

continues

Table B-18 Operator Precedence and Associativity (*Continued*)

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>||</td><td>Left to right</td></tr>
<tr><td>?:</td><td>Right to left</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>Right to left</td></tr>
<tr><td>,</td><td>Left to right</td></tr>
</table>

The table entries are in order from highest precedence to lowest precedence.

The comma `(,)` operator listed in the table is for compatibility with the ANSI-C comma operator, which can be used to evaluate a set of expressions in left-to-right order and return the value of the rightmost expression. This operator is provided strictly for compatibility with C and should generally not be used.

The `()` entry in the table of operator precedence represents a function call. The `[]` entry in the table of operator precedence represents an array or associative array reference.

C

Provider Arguments Reference

This appendix shows the providers available on Solaris Nevada, circa May 2010 (which contains the most comprehensive collection of providers to date). The "Providers" section summarizes the probes for each provider and the argument types. Refer to the provider chapters in the DTrace Guide for the full reference for each provider, which includes an explanation for the individual arguments. The "Arguments" section summarizes some common argument types.

Providers

Tables C-1 through C-15 cover all the providers.

Table C-1 fc Provider Probes and Arguments

Probe	Arguments
`fc:::abts-receive`	`conninfo_t *, fc_port_info_t *, fc_port_info_t *`
`fc:::fabric-login-end`	`conninfo_t *, fc_port_info_t *`
`fc:::fabric-login-start`	`conninfo_t *, fc_port_info_t *`
`fc:::link-down`	`conninfo_t *`

continues

Table C-1 fc Provider Probes and Arguments (*Continued*)

Probe	Arguments
fc:::link-up	conninfo_t *
fc:::rport-login-end	conninfo_t *, fc_port_info_t *, fc_port_info_t *, int, int
fc:::rport-login-start	conninfo_t *, fc_port_info_t *, fc_port_info_t *, int
fc:::rport-logout-end	conninfo_t *, fc_port_info_t *, fc_port_info_t *, int
fc:::rport-logout-start	conninfo_t *, fc_port_info_t *, fc_port_info_t *, int
fc:::rscn-receive	conninfo_t *, int
fc:::scsi-command	conninfo_t *, fc_port_info_t *, scsicmd_t *, fc_port_info_t *
fc:::scsi-response	conninfo_t *, fc_port_info_t *, scsicmd_t *, fc_port_info_t *
fc:::xfer-done	conninfo_t *, fc_port_info_t *, scsicmd_t *, fc_port_info_t *, fc_xferinfo_t *
fc:::xfer-start	conninfo_t *, fc_port_info_t *, scsicmd_t *, fc_port_info_t *, fc_xferinfo_t *

Table C-2 fsinfo Provider Probes and Arguments

Probe	Arguments
fsinfo:::*	fileinfo_t *, int

Table C-3 io Provider Probes and Arguments

Probe	Arguments
io:::done	bufinfo_t *, devinfo_t *, fileinfo_t *
io:::start	bufinfo_t *, devinfo_t *, fileinfo_t *
io:::wait-done	bufinfo_t *, devinfo_t *, fileinfo_t *
io:::wait-start	bufinfo_t *, devinfo_t *, fileinfo_t *

Table C-4 ip Provider Probes and Arguments

Probe	Arguments
ip:::receive	pktinfo_t *, csinfo_t *, ipinfo_t *, ifinfo_t *, ipv4info_t *, ipv6info_t *
ip:::send	pktinfo_t *, csinfo_t *, ipinfo_t *, ifinfo_t *, ipv4info_t *, ipv6info_t *

Table C-5 iscsi Provider Probes and Arguments

Probe	Arguments
iscsi:::async-send	conninfo_t *, iscsiinfo_t *
iscsi:::data-receive	conninfo_t *, iscsiinfo_t *
iscsi:::data-request	conninfo_t *, iscsiinfo_t *
iscsi:::data-send	conninfo_t *, iscsiinfo_t *
iscsi:::login-command	conninfo_t *, iscsiinfo_t *
iscsi:::login-response	conninfo_t *, iscsiinfo_t *
iscsi:::logout-command	conninfo_t *, iscsiinfo_t *
iscsi:::logout-response	conninfo_t *, iscsiinfo_t *
iscsi:::nop-receive	conninfo_t *, iscsiinfo_t *
iscsi:::nop-send	conninfo_t *, iscsiinfo_t *
iscsi:::scsi-command	conninfo_t *, iscsiinfo_t *, scsicmd_t *
iscsi:::scsi-response	conninfo_t *, iscsiinfo_t *
iscsi:::task-command	conninfo_t *, iscsiinfo_t *
iscsi:::task-response	conninfo_t *, iscsiinfo_t *
iscsi:::text-command	conninfo_t *, iscsiinfo_t *
iscsi:::text-response	conninfo_t *, iscsiinfo_t *
iscsi:::xfer-done	conninfo_t *, iscsiinfo_t *, xferinfo_t *, uint32_t, uintptr_t, uint32_t, uint32_t, uint32_t, int
iscsi:::xfer-start	conninfo_t *, iscsiinfo_t *, xferinfo_t *, uint32_t, uintptr_t, uint32_t, uint32_t, uint32_t, int

Table C-6 mib Provider Probes and Arguments

Probe	Arguments
mib:::	int

Table C-7 nfsv3 Provider Probes and Arguments

Probe	Arguments
nfsv3:::op-access-done	conninfo_t *, nfsv3opinfo_t *, ACCESS3res *
nfsv3:::op-access-start	conninfo_t *, nfsv3opinfo_t *, ACCESS3args *
nfsv3:::op-commit-done	conninfo_t *, nfsv3opinfo_t *, COMMIT3res *
nfsv3:::op-commit-start	conninfo_t *, nfsv3opinfo_t *, COMMIT3args *
nfsv3:::op-create-done	conninfo_t *, nfsv3opinfo_t *, CREATE3res *
nfsv3:::op-create-start	conninfo_t *, nfsv3opinfo_t *, CREATE3args *
nfsv3:::op-fsinfo-done	conninfo_t *, nfsv3opinfo_t *, FSINFO3res *
nfsv3:::op-fsinfo-start	conninfo_t *, nfsv3opinfo_t *, FSINFO3args *
nfsv3:::op-fsstat-done	conninfo_t *, nfsv3opinfo_t *, FSSTAT3res *
nfsv3:::op-fsstat-start	conninfo_t *, nfsv3opinfo_t *, FSSTAT3args *
nfsv3:::op-getattr-done	conninfo_t *, nfsv3opinfo_t *, GETATTR3res *
nfsv3:::op-getattr-start	conninfo_t *, nfsv3opinfo_t *, GETATTR3args *
nfsv3:::op-link-done	conninfo_t *, nfsv3opinfo_t *, LINK3res *
nfsv3:::op-link-start	conninfo_t *, nfsv3opinfo_t *, LINK3args *
nfsv3:::op-lookup-done	conninfo_t *, nfsv3opinfo_t *, LOOKUP3res *
nfsv3:::op-lookup-start	conninfo_t *, nfsv3opinfo_t *, LOOKUP3args *
nfsv3:::op-mkdir-done	conninfo_t *, nfsv3opinfo_t *, MKDIR3res *
nfsv3:::op-mkdir-start	conninfo_t *, nfsv3opinfo_t *, MKDIR3args *
nfsv3:::op-mknod-done	conninfo_t *, nfsv3opinfo_t *, MKNOD3res *

Table C-7 nfsv3 Provider Probes and Arguments (*Continued*)

Probe	Arguments
nfsv3:::op-mknod-start	conninfo_t *, nfsv3opinfo_t *, MKNOD3args *
nfsv3:::op-null-done	conninfo_t *, nfsv3opinfo_t *
nfsv3:::op-null-start	conninfo_t *, nfsv3opinfo_t *
nfsv3:::op-pathconf-done	conninfo_t *, nfsv3opinfo_t *, PATHCONF3res *
nfsv3:::op-pathconf-start	conninfo_t *, nfsv3opinfo_t *, PATHCONF3args *
nfsv3:::op-read-done	conninfo_t *, nfsv3opinfo_t *, READ3res *
nfsv3:::op-read-start	conninfo_t *, nfsv3opinfo_t *, READ3args *
nfsv3:::op-readdir-done	conninfo_t *, nfsv3opinfo_t *, READDIR3res *
nfsv3:::op-readdir-start	conninfo_t *, nfsv3opinfo_t *, READDIR3args *
nfsv3:::op-readdirplus-done	conninfo_t *, nfsv3opinfo_t *, READDIRPLUS3res *
nfsv3:::op-readdirplus-start	conninfo_t *, nfsv3opinfo_t *, READDIRPLUS3args *
nfsv3:::op-readlink-done	conninfo_t *, nfsv3opinfo_t *, READLINK3res *
nfsv3:::op-readlink-start	conninfo_t *, nfsv3opinfo_t *, READLINK3args *
nfsv3:::op-remove-done	conninfo_t *, nfsv3opinfo_t *, REMOVE3res *
nfsv3:::op-remove-start	conninfo_t *, nfsv3opinfo_t *, REMOVE3args *
nfsv3:::op-rename-done	conninfo_t *, nfsv3opinfo_t *, RENAME3res *
nfsv3:::op-rename-start	conninfo_t *, nfsv3opinfo_t *, RENAME3args *
nfsv3:::op-rmdir-done	conninfo_t *, nfsv3opinfo_t *, RMDIR3res *
nfsv3:::op-rmdir-start	conninfo_t *, nfsv3opinfo_t *, RMDIR3args *

continues

Table C-7 nfsv3 Provider Probes and Arguments (*Continued*)

Probe	Arguments
`nfsv3:::op-setattr-done`	`conninfo_t *, nfsv3opinfo_t *, SETATTR3res *`
`nfsv3:::op-setattr-start`	`conninfo_t *, nfsv3opinfo_t *, SETATTR3args *`
`nfsv3:::op-symlink-done`	`conninfo_t *, nfsv3opinfo_t *, SYMLINK3res *`
`nfsv3:::op-symlink-start`	`conninfo_t *, nfsv3opinfo_t *, SYMLINK3args *`
`nfsv3:::op-write-done`	`conninfo_t *, nfsv3opinfo_t *, WRITE3res *`
`nfsv3:::op-write-start`	`conninfo_t *, nfsv3opinfo_t *, WRITE3args *`

Table C-8 nfsv4 Provider Probes and Arguments

Probe	Arguments
`nfsv4:::cb-recall-done`	`conninfo_t *, nfsv4cbinfo_t *, CB_RECALL4res *`
`nfsv4:::cb-recall-start`	`conninfo_t *, nfsv4cbinfo_t *, CB_RECALL4args *`
`nfsv4:::compound-done`	`conninfo_t *, nfsv4opinfo_t *, COMPOUND4res *`
`nfsv4:::compound-start`	`conninfo_t *, nfsv4opinfo_t *, COMPOUND4args *`
`nfsv4:::null-done`	`conninfo_t *`
`nfsv4:::null-start`	`conninfo_t *`
`nfsv4:::op-access-done`	`conninfo_t *, nfsv4opinfo_t *, ACCESS4res *`
`nfsv4:::op-access-start`	`conninfo_t *, nfsv4opinfo_t *, ACCESS4args *`
`nfsv4:::op-close-done`	`conninfo_t *, nfsv4opinfo_t *, CLOSE4res *`
`nfsv4:::op-close-start`	`conninfo_t *, nfsv4opinfo_t *, CLOSE4args *`
`nfsv4:::op-commit-done`	`conninfo_t *, nfsv4opinfo_t *, COMMIT4res *`

Table C-8 nfsv4 Provider Probes and Arguments (*Continued*)

Probe	Arguments
nfsv4:::op-commit-start	conninfo_t *, nfsv4opinfo_t *, COMMIT4args *
nfsv4:::op-create-done	conninfo_t *, nfsv4opinfo_t *, CREATE4res *
nfsv4:::op-create-start	conninfo_t *, nfsv4opinfo_t *, CREATE4args *
nfsv4:::op-delegpurge-done	conninfo_t *, nfsv4opinfo_t *, DELEGPURGE4res *
nfsv4:::op-delegpurge-start	conninfo_t *, nfsv4opinfo_t *, DELEGPURGE4args *
nfsv4:::op-delegreturn-done	conninfo_t *, nfsv4opinfo_t *, DELEGRETURN4res *
nfsv4:::op-delegreturn-start	conninfo_t *, nfsv4opinfo_t *, DELEGRETURN4args *
nfsv4:::op-getattr-done	conninfo_t *, nfsv4opinfo_t *, GETATTR4res *
nfsv4:::op-getattr-start	conninfo_t *, nfsv4opinfo_t *, GETATTR4args *
nfsv4:::op-getfh-done	conninfo_t *, nfsv4opinfo_t *, GETFH4res *
nfsv4:::op-getfh-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-link-done	conninfo_t *, nfsv4opinfo_t *, LINK4res *
nfsv4:::op-link-start	conninfo_t *, nfsv4opinfo_t *, LINK4args *
nfsv4:::op-lock-done	conninfo_t *, nfsv4opinfo_t *, LOCK4res *
nfsv4:::op-lock-start	conninfo_t *, nfsv4opinfo_t *, LOCK4args *
nfsv4:::op-lockt-done	conninfo_t *, nfsv4opinfo_t *, LOCKT4res *
nfsv4:::op-lockt-start	conninfo_t *, nfsv4opinfo_t *, LOCKT4args *
nfsv4:::op-locku-done	conninfo_t *, nfsv4opinfo_t *, LOCKU4res *
nfsv4:::op-locku-start	conninfo_t *, nfsv4opinfo_t *, LOCKU4args *

continues

Table C-8 nfsv4 Provider Probes and Arguments (*Continued*)

Probe	Arguments
nfsv4:::op-lookup-done	conninfo_t *, nfsv4opinfo_t *, LOOKUP4res *
nfsv4:::op-lookup-start	conninfo_t *, nfsv4opinfo_t *, LOOKUP4args *
nfsv4:::op-lookupp-done	conninfo_t *, nfsv4opinfo_t *, LOOKUPP4res *
nfsv4:::op-lookupp-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-nverify-done	conninfo_t *, nfsv4opinfo_t *, NVERIFY4res *
nfsv4:::op-nverify-start	conninfo_t *, nfsv4opinfo_t *, NVERIFY4args *
nfsv4:::op-open-confirm-done	conninfo_t *, nfsv4opinfo_t *, OPEN_CONFIRM4res *
nfsv4:::op-open-confirm-start	conninfo_t *, nfsv4opinfo_t *, OPEN_CONFIRM4args *
nfsv4:::op-open-done	conninfo_t *, nfsv4opinfo_t *, OPEN4res *
nfsv4:::op-open-downgrade-done	conninfo_t *, nfsv4opinfo_t *, OPEN_DOWNGRADE4res *
nfsv4:::op-open-downgrade-start	conninfo_t *, nfsv4opinfo_t *, OPEN_DOWNGRADE4args *
nfsv4:::op-open-start	conninfo_t *, nfsv4opinfo_t *, OPEN4args *
nfsv4:::op-openattr-done	conninfo_t *, nfsv4opinfo_t *, OPENATTR4res *
nfsv4:::op-openattr-start	conninfo_t *, nfsv4opinfo_t *, OPENATTR4args *
nfsv4:::op-putfh-done	conninfo_t *, nfsv4opinfo_t *, PUTFH4res *
nfsv4:::op-putfh-start	conninfo_t *, nfsv4opinfo_t *, PUTFH4args *
nfsv4:::op-putpubfh-done	conninfo_t *, nfsv4opinfo_t *, PUTPUBFH4res *
nfsv4:::op-putpubfh-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-putrootfh-done	conninfo_t *, nfsv4opinfo_t *, PUTROOTFH4res *
nfsv4:::op-putrootfh-start	conninfo_t *, nfsv4opinfo_t *

Table C-8 nfsv4 Provider Probes and Arguments (*Continued*)

Probe	Arguments
nfsv4:::op-read-done	conninfo_t *, nfsv4opinfo_t *, READ4res *
nfsv4:::op-read-start	conninfo_t *, nfsv4opinfo_t *, READ4args *
nfsv4:::op-readdir-done	conninfo_t *, nfsv4opinfo_t *, READDIR4res *
nfsv4:::op-readdir-start	conninfo_t *, nfsv4opinfo_t *, READDIR4args *
nfsv4:::op-readlink-done	conninfo_t *, nfsv4opinfo_t *, READLINK4res *
nfsv4:::op-readlink-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-release-lockowner-done	conninfo_t *, nfsv4opinfo_t *, RELEASE_LOCKOWNER4res *
nfsv4:::op-release-lockowner-start	conninfo_t *, nfsv4opinfo_t *, RELEASE_LOCKOWNER4args *
nfsv4:::op-remove-done	conninfo_t *, nfsv4opinfo_t *, REMOVE4res *
nfsv4:::op-remove-start	conninfo_t *, nfsv4opinfo_t *, REMOVE4args *
nfsv4:::op-rename-done	conninfo_t *, nfsv4opinfo_t *, RENAME4res *
nfsv4:::op-rename-start	conninfo_t *, nfsv4opinfo_t *, RENAME4args *
nfsv4:::op-renew-done	conninfo_t *, nfsv4opinfo_t *, RENEW4res *
nfsv4:::op-renew-start	conninfo_t *, nfsv4opinfo_t *, RENEW4args *
nfsv4:::op-restorefh-done	conninfo_t *, nfsv4opinfo_t *, RESTOREFH4res *
nfsv4:::op-restorefh-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-savefh-done	conninfo_t *, nfsv4opinfo_t *, SAVEFH4res *
nfsv4:::op-savefh-start	conninfo_t *, nfsv4opinfo_t *
nfsv4:::op-secinfo-done	conninfo_t *, nfsv4opinfo_t *, SECINFO4res *
nfsv4:::op-secinfo-start	conninfo_t *, nfsv4opinfo_t *, SECINFO4args *

continues

Table C-8 nfsv4 Provider Probes and Arguments (*Continued*)

Probe	Arguments
nfsv4:::op-setattr-done	conninfo_t *, nfsv4opinfo_t *, SETATTR4res *
nfsv4:::op-setattr-start	conninfo_t *, nfsv4opinfo_t *, SETATTR4args *
nfsv4:::op-setclientid-confirm-done	conninfo_t *, nfsv4opinfo_t *, SETCLIENTID_CONFIRM4res *
nfsv4:::op-setclientid-confirm-start	conninfo_t *, nfsv4opinfo_t *, SETCLIENTID_CONFIRM4args *
nfsv4:::op-setclientid-done	conninfo_t *, nfsv4opinfo_t *, SETCLIENTID4res *
nfsv4:::op-setclientid-start	conninfo_t *, nfsv4opinfo_t *, SETCLIENTID4args *
nfsv4:::op-verify-done	conninfo_t *, nfsv4opinfo_t *, VERIFY4res *
nfsv4:::op-verify-start	conninfo_t *, nfsv4opinfo_t *, VERIFY4args *
nfsv4:::op-write-done	conninfo_t *, nfsv4opinfo_t *, WRITE4res *
nfsv4:::op-write-start	conninfo_t *, nfsv4opinfo_t *, WRITE4args *

Table C-9 proc Provider Probes and Arguments

Probe	Arguments
proc:::create	psinfo_t *
proc:::exec	string
proc:::exec-failure	int
proc:::exec-success	
proc:::exit	int
proc:::fault	int, siginfo_t *
proc:::lwp-create	lwpsinfo_t *, psinfo_t *
proc:::lwp-exit	
proc:::signal-clear	int, siginfo_t *
proc:::signal-discard	lwpsinfo_t *, psinfo_t *, int
proc:::signal-handle	int, siginfo_t *, int (*)()
proc:::signal-send	lwpsinfo_t *, psinfo_t *, int

Table C-10 sched Provider Probes and Arguments

Probe	Arguments
sched:::change-pri	lwpsinfo_t *, psinfo_t *, pri_t
sched:::cpucaps-sleep	lwpsinfo_t *, psinfo_t *
sched:::cpucaps-wakeup	lwpsinfo_t *, psinfo_t *
sched:::dequeue	lwpsinfo_t *, psinfo_t *, cpuinfo_t *
sched:::enqueue	lwpsinfo_t *, psinfo_t *, cpuinfo_t *, int
sched:::off-cpu	lwpsinfo_t *, psinfo_t *
sched:::schedctl-nopreempt	lwpsinfo_t *, psinfo_t *, int
sched:::schedctl-preempt	lwpsinfo_t *, psinfo_t *
sched:::schedctl-yield	int
sched:::surrender	lwpsinfo_t *, psinfo_t *
sched:::tick	lwpsinfo_t *, psinfo_t *
sched:::wakeup	lwpsinfo_t *, psinfo_t *

Table C-11 srp Provider Probes and Arguments

Probe	Arguments
srp:::login-command	conninfo_t *, srp_portinfo_t *, srp_logininfo_t *
srp:::login-response	conninfo_t *, srp_portinfo_t *, srp_logininfo_t *
srp:::logout-command	conninfo_t *, srp_portinfo_t *
srp:::scsi-command	conninfo_t *, srp_portinfo_t *, scsicmd_t *, srp_taskinfo_t *
srp:::scsi-response	conninfo_t *, srp_portinfo_t *, srp_taskinfo_t *
srp:::service-down	conninfo_t *, srp_portinfo_t *
srp:::service-up	conninfo_t *, srp_portinfo_t *
srp:::task-command	conninfo_t *, srp_portinfo_t *, srp_taskinfo_t *
srp:::task-response	conninfo_t *, srp_portinfo_t *, srp_taskinfo_t *
srp:::xfer-done	conninfo_t *, srp_portinfo_t *, xferinfo_t *, srp_taskinfo_t *
srp:::xfer-start	conninfo_t *, srp_portinfo_t *, xferinfo_t *, srp_taskinfo_t *

Table C-12 sysevent Provider Probes and Arguments

Probe	Arguments
sysevent:::post	syseventchaninfo_t *, syseventinfo_t *

Table C-13 tcp Provider Probes and Arguments

Probe	Arguments
tcp:::accept-established	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::accept-refused	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::connect-established	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::connect-refused	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::connect-request	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::receive	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::send	pktinfo_t *, csinfo_t *, ipinfo_t *, tcpsinfo_t *, tcpinfo_t *
tcp:::state-change	void, csinfo_t *, void, tcpsinfo_t *, void, tcplsinfo_t *

Table C-14 udp Provider Probes and Arguments

Probe	Arguments
udp:::receive	pktinfo_t *, csinfo_t *, ipinfo_t *, udpsinfo_t *, udpinfo_t *
udp:::send	pktinfo_t *, csinfo_t *, ipinfo_t *, udpsinfo_t *, udpinfo_t *

Table C-15 xpv Provider Probes and Arguments

Probe	Arguments
xpv:::add-to-physmap-end	int
xpv:::add-to-physmap-start	domid_t, uint_t, ulong_t, ulong_t
xpv:::decrease-reservation-end	int
xpv:::decrease-reservation-start	domid_t, ulong_t, uint_t, ulong_t *
xpv:::dom-create-end	int
xpv:::dom-create-start	xen_domctl_t *
xpv:::dom-destroy-end	int
xpv:::dom-destroy-start	domid_t
xpv:::dom-pause-end	int
xpv:::dom-pause-start	domid_t
xpv:::dom-unpause-end	int
xpv:::dom-unpause-start	domid_t
xpv:::evtchn-op-end	int
xpv:::evtchn-op-start	int, void *
xpv:::increase-reservation-end	int
xpv:::increase-reservation-start	domid_t, ulong_t, uint_t, ulong_t *
xpv:::mmap-end	int
xpv:::mmap-entry	ulong_t, ulong_t, ulong_t
xpv:::mmap-start	domid_t, int, privcmd_mmap_entry_t *
xpv:::mmapbatch-end	int, struct seg *, caddr_t
xpv:::mmapbatch-start	domid_t, int, caddr_t
xpv:::mmu-ext-op-end	int
xpv:::mmu-ext-op-start	int, int, struct mmuext_op *
xpv:::mmu-update-end	int
xpv:::mmu-update-start	int, int, mmu_update_t *
xpv:::populate-physmap-end	int
xpv:::populate-physmap-start	domid_t, ulong_t, ulong_t *
xpv:::set-memory-map-end	int
xpv:::set-memory-map-start	domid_t, int, struct xen_memory_map *
xpv:::setvcpucontext-end	int
xpv:::setvcpucontext-start	domid_t, vcpu_guest_context_t *

Arguments

Common argument types are specified in this section. These are from the Solaris Nevada translator files in /usr/lib/dtrace and also documented in the provider chapters of the DTrace Guide.

bufinfo_t

```
typedef struct bufinfo {
      int b_flags;              /* buffer status */
      size_t b_bcount;          /* number of bytes */
      caddr_t b_addr;           /* buffer address */
      uint64_t b_lblkno;        /* block # on device */
      uint64_t b_blkno;         /* expanded block # on device */
      size_t b_resid;           /* # of bytes not transferred */
      size_t b_bufsize;         /* size of allocated buffer */
      caddr_t b_iodone;         /* I/O completion routine */
      int b_error;              /* expanded error field */
      dev_t b_edev;             /* extended device */
} bufinfo_t;
```

devinfo_t

```
typedef struct devinfo {
      int dev_major;            /* major number */
      int dev_minor;            /* minor number */
      int dev_instance;         /* instance number */
      string dev_name;          /* name of device */
      string dev_statname;      /* name of device + instance/minor */
      string dev_pathname;      /* pathname of device */
} devinfo_t;
```

fileinfo_t

```
typedef struct fileinfo {
      string fi_name;           /* name (basename of fi_pathname) */
      string fi_dirname;        /* directory (dirname of fi_pathname) */
      string fi_pathname;       /* full pathname */
      offset_t fi_offset;       /* offset within file */
      string fi_fs;             /* filesystem */
      string fi_mount;          /* mount point of file system */
      int fi_oflags;            /* open(2) flags for file descriptor */
} fileinfo_t;
```

cpuinfo_t

```
typedef struct cpuinfo {
        processorid_t cpu_id;            /* CPU identifier */
        psetid_t cpu_pset;               /* processor set identifier */
        chipid_t cpu_chip;               /* chip identifier */
        lgrp_id_t cpu_lgrp;              /* locality group identifer */
        processor_info_t cpu_info;       /* CPU information */
} cpuinfo_t;
```

lwpsinfo_t

```
typedef struct lwpsinfo {
        int pr_flag;                /* flags; see below */
        id_t pr_lwpid;              /* LWP id */
        uintptr_t pr_addr;          /* internal address of thread */
        uintptr_t pr_wchan;         /* wait addr for sleeping thread */
        char pr_stype;              /* synchronization event type */
        char pr_state;              /* numeric thread state */
        char pr_sname;              /* printable character for pr_state */
        char pr_nice;               /* nice for cpu usage */
        short pr_syscall;           /* system call number (if in syscall) */
        int pr_pri;                 /* priority,  high value = high priority */
        char pr_clname[PRCLSZ];     /* scheduling class name */
        processorid_t pr_onpro;     /* processor which last ran this thread */
        processorid_t pr_bindpro; /* processor to which thread is bound */
        psetid_t pr_bindpset;       /* processor set to which thread is bound */
} lwpsinfo_t;
```

psinfo_t

```
typedef struct psinfo {
        int       pr_nlwp;            /* number of active lwps in the process */
        pid_t     pr_pid;             /* unique process id */
        pid_t     pr_ppid;            /* process id of parent */
        pid_t     pr_pgid;            /* pid of process group leader */
        pid_t     pr_sid;             /* session id */
        uid_t     pr_uid;             /* real user id */
        uid_t     pr_euid;            /* effective user id */
        gid_t     pr_gid;             /* real group id */
        gid_t     pr_egid;            /* effective group id */
        uintptr_t pr_addr;            /* address of process */
        dev_t     pr_ttydev;          /* controlling tty device (or PRNODEV) */
        timestruc_t pr_start;         /* process start time,  from the epoch */
        char      pr_fname[PRFNSZ];   /* name of execed file */
        char      pr_psargs[PRARGSZ]; /* initial characters of arg list */
        int       pr_argc;            /* initial argument count */
        uintptr_t pr_argv;            /* address of initial argument vector */
        uintptr_t pr_envp;            /* address of initial environment vector */
        char      pr_dmodel;          /* data model of the process */
        taskid_t pr_taskid;           /* task id */
        projid_t pr_projid;           /* project id */
        poolid_t pr_poolid;           /* pool id */
        zoneid_t pr_zoneid;           /* zone id */
} psinfo_t;
```

conninfo_t

```
/*
 * The conninfo_t structure should be used by all application protocol
 * providers as the first arguments to indicate some basic information
 * about the connection. This structure may be augmented to accommodate
 * the particularities of additional protocols in the future.
 */
typedef struct conninfo {
      string ci_local;        /* local host address */
      string ci_remote;       /* remote host address */
      string ci_protocol;     /* protocol (ipv4,  ipv6,  etc) */
} conninfo_t;
```

pktinfo_t

```
/*
 * pktinfo is where packet ID info can be made available for deeper
 * analysis if packet IDs become supported by the kernel in the future.
 * The pkt_addr member is currently always NULL.
 */
typedef struct pktinfo {
      uintptr_t pkt_addr;
} pktinfo_t;
```

csinfo_t

```
/*
 * csinfo is where connection state info is made available.
 */
typedef struct csinfo {
      uintptr_t cs_addr;
      uint64_t cs_cid;
      pid_t cs_pid;
      zoneid_t cs_zoneid;
} csinfo_t;
```

ipinfo_t

```
/*
 * ipinfo contains common IP info for both IPv4 and IPv6.
 */
typedef struct ipinfo {
      uint8_t ip_ver;         /* IP version (4,  6) */
      uint32_t ip_plength;    /* payload length */
      string ip_saddr;        /* source address */
      string ip_daddr;        /* destination address */
} ipinfo_t;
```

ifinfo_t

```
/*
 * ifinfo contains network interface info.
 */
typedef struct ifinfo {
      string if_name;                /* interface name */
      int8_t if_local;               /* is delivered locally */
      netstackid_t if_ipstack;       /* ipstack ID */
      uintptr_t if_addr;             /* pointer to raw ill_t */
} ifinfo_t;
```

ipv4info_t

```
/*
 * ipv4info is a translated version of the IPv4 header (with raw pointer).
 * These values are NULL if the packet is not IPv4.
 */
typedef struct ipv4info {
      uint8_t ipv4_ver;         /* IP version (4) */
      uint8_t ipv4_ihl;         /* header length,  bytes */
      uint8_t ipv4_tos;         /* type of service field */
      uint16_t ipv4_length;     /* length (header + payload) */
      uint16_t ipv4_ident;      /* identification */
      uint8_t ipv4_flags;       /* IP flags */
      uint16_t ipv4_offset;     /* fragment offset */
      uint8_t ipv4_ttl;         /* time to live */
      uint8_t ipv4_protocol;    /* next level protocol */
      string ipv4_protostr;     /* next level protocol,  as a string */
      uint16_t ipv4_checksum;   /* header checksum */
      ipaddr_t ipv4_src;        /* source address */
      ipaddr_t ipv4_dst;        /* destination address */
      string ipv4_saddr;        /* source address,  string */
      string ipv4_daddr;        /* destination address,  string */
      ipha_t *ipv4_hdr;         /* pointer to raw header */
} ipv4info_t;
```

ipv6info_t

```
/*
 * ipv6info is a translated version of the IPv6 header (with raw pointer).
 * These values are NULL if the packet is not IPv6.
 */
typedef struct ipv6info {
      uint8_t ipv6_ver;         /* IP version (6) */
      uint8_t ipv6_tclass;      /* traffic class */
      uint32_t ipv6_flow;       /* flow label */
      uint16_t ipv6_plen;       /* payload length */
      uint8_t ipv6_nexthdr;     /* next header protocol */
      string ipv6_nextstr;      /* next header protocol,  as a string */
      uint8_t ipv6_hlim;        /* hop limit */
      in6_addr_t *ipv6_src;     /* source address */
      in6_addr_t *ipv6_dst;     /* destination address */
```

continues

```
      string ipv6_saddr;      /* source address,  string */
      string ipv6_daddr;      /* destination address,  string */
      ip6_t *ipv6_hdr;        /* pointer to raw header */
} ipv6info_t;
```

tcpinfo_t

```
/*
 * tcpinfo is the TCP header fields.
 */
typedef struct tcpinfo {
      uint16_t tcp_sport;     /* source port */
      uint16_t tcp_dport;     /* destination port */
      uint32_t tcp_seq;       /* sequence number */
      uint32_t tcp_ack;       /* acknowledgment number */
      uint8_t tcp_offset;     /* data offset,  in bytes */
      uint8_t tcp_flags;      /* flags */
      uint16_t tcp_window;    /* window size */
      uint16_t tcp_checksum;  /* checksum */
      uint16_t tcp_urgent;    /* urgent data pointer */
      tcph_t *tcp_hdr;        /* raw TCP header */
} tcpinfo_t;
```

tcpsinfo_t

```
/*
 * tcpsinfo contains stable TCP details from tcp_t.
 */
typedef struct tcpsinfo {
      uintptr_t tcps_addr;
      int tcps_local;         /* is delivered locally,  boolean */
      int tcps_active;        /* active open (from here),  boolean */
      uint16_t tcps_lport;          /* local port */
      uint16_t tcps_rport;          /* remote port */
      string tcps_laddr;            /* local address,  as a string */
      string tcps_raddr;            /* remote address,  as a string */
      int32_t tcps_state;           /* TCP state */
      uint32_t tcps_iss;            /* Initial sequence # sent */
      uint32_t tcps_suna;           /* sequence # sent but unacked */
      uint32_t tcps_snxt;           /* next sequence # to send */
      uint32_t tcps_rack;           /* sequence # we have acked */
      uint32_t tcps_rnxt;           /* next sequence # expected */
      uint32_t tcps_swnd;           /* send window size */
      int32_t tcps_snd_ws;          /* send window scaling */
      uint32_t tcps_rwnd;           /* receive window size */
      int32_t tcps_rcv_ws;          /* receive window scaling */
      uint32_t tcps_cwnd;           /* congestion window */
      uint32_t tcps_cwnd_ssthresh;  /* threshold for congestion avoidance */
      uint32_t tcps_sack_fack;      /* SACK sequence # we have acked */
      uint32_t tcps_sack_snxt;      /* next SACK seq # for retransmission */
      uint32_t tcps_rto;            /* round-trip timeout,  msec */
      uint32_t tcps_mss;            /* max segment size */
      int tcps_retransmit;          /* retransmit send event,  boolean */
} tcpsinfo_t;
```

tcplsinfo_t

```
/*
 * tcplsinfo provides the old tcp state for state changes.
 */
typedef struct tcplsinfo {
      int32_t tcps_state;           /* previous TCP state */
} tcplsinfo_t;
```

This page intentionally left blank

D

DTrace on FreeBSD

This appendix covers enabling DTrace on FreeBSD and includes an e-mail from John Birrell.

Enabling DTrace on FreeBSD 7.1 and 8.0

DTrace is not available in FreeBSD 7.1/8.0 following an installation. A kernel build is required after editing the kernel configuration file. The DTrace modules must also explicitly be loaded after the new kernel is booted.

You can reference the online FreeBSD documentation for enabling DTrace here:

www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/dtrace-enable.html

Information on the kernel configuration file and building kernels can be found here:

www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig-building.html

www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig-config.html

For your convenience, the steps are outlined here:

1. `cd` to `/usr/src/sys/i386/conf`.

   This is the directory location of the kernel configuration file, `GENERIC`.

   Run `cp GENERIC GENERIC_DTRACE`.

   It's a good idea to keep the original file intact and make a copy for editing.
2. Run `edit GENERIC_DTRACE`, adding these two options:

   `options KDTRACE_HOOKS`

   `options DDB_CTF`

> **Note**
>
> From the documentation, if you're building on AMD64, you also need to add this:
>
> ```
> options KDTRACE_FRAME
> ```

> **Note**
>
> `options KDTRACE_HOOKS` was already in the configuration file but commented out. The `options DDB_CTF` needed to be explicitly added.

   Run `cd /usr/src`.

   Run `make WITH_CTF=1 KERNCONF=GENERIC_DTRACE kernel`.

   Build a kernel with the `GENERIC_DTRACE` configuration file and `WITH_CTF=1`.

   When the build completes, reboot.
3. Type the following command after reboot to load the dtrace kernel modules:

   `kldload dtraceall`

At this point, DTrace is ready to use on your FreeBSD 7.1 system!

The following example shows the `dtrace(1)` command with the `-l` flag to list all probes, getting a total number of probes (33,207) on our FreeBSD system. We can see that FreeBSD currently has a limited number of providers available: dtrace, dtmalloc, fbt, proc, syscall, and profile.

```
freebsd# dtrace -l | wc -l
   33207
freebsd# dtrace -l | awk '{print $2}' | uniq
PROVIDER
dtrace
```

```
dtmalloc
fbt
proc
syscall
profile
```

DTrace for FreeBSD: John Birrell

The port of DTrace to FreeBSD was performed by John Birrell, a FreeBSD contributor who was based on the Victorian coast, Australia. His e-mail to `freebsd-current@freebsd.org` announcing DTrace for FreeBSD in May 2006 has been reproduced here, in memory of a remarkably talented and determined software engineer.

```
Date: Wed May 24 23:55:12 PDT 2006
From: John Birrell <jb@what-creek.com>
Subject: DTrace for FreeBSD - Status Update

It's nearly 8 weeks since I started porting DTrace to FreeBSD and I
thought I would post a status update including today's significant
emotional event. 8-)

For those who don't know what DTrace is or which company designed it,
here are a few links:

The BigAdmin: <http://www.sun.com/bigadmin/content/dtrace/>
A Blurb: <http://www.sun.com/2004-0518/feature/index.html>
The Guide: <http://docs.sun.com/app/docs/doc/817-6223>
My FreeBSD Project Page: <http://people.freebsd.org/~jb/dtrace/index.html>

Much of the basic DTrace infrastructure is in place now. Of the 1039
DTrace tests that Sun runs on Solaris, 793 now pass on FreeBSD.

We've got the following providers:

- dtrace
- profile
- syscall
- sdt
- fbt

As of today, loading those providers on a GENERIC kernel gives 32,519
probes.

Today's significant emotional event added over 30,000 of those, thanks
to the Function Boundary Tracing (fbt) provider. It provides the
instrumentation of the entry and return of every (non-leaf) function
in the kernel and (non-DTrace provider) modules.
```

```
Here is an example of what fbt can do.... The following script creates
a probe on the entry to the kernel malloc() function. It dereferences
the second argument to the malloc_type structure and then quantizes the
size of the mallocs being made according to the malloc type name.

The script:

fbt:kernel:malloc:entry
{
        mt = (struct malloc_type *) arg1;
        @[stringof(mt->ks_shortdesc)] = quantize(arg0)
}


The output:


  vmem
           value  ------------- Distribution ------------- count
               2 |                                         0
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 56
               8 |                                         0

[...]

  nfsserver_srvdesc
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@@@@@@@@                     8991
              16 |                                         0
              32 |                                         0
              64 |                                         0
             128 |@@@@@@@@@@@@@@@@@@@@                     8991
             256 |                                         0

  temp
           value  ------------- Distribution ------------- count
               4 |                                         0
               8 |@@@@@@@@@@@@@                            935
              16 |@@                                       151
              32 |@@@                                      184
              64 |@                                        66
             128 |@                                        97
             256 |                                         30
             512 |                                         22
            1024 |                                         13
            2048 |                                         4
            4096 |                                         28
            8192 |@@@@@@@@@@@@@@@@@@@                      1359
           16384 |                                         0
```

```
  dtrace
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@                                        23
               2 |                                         19
               4 |@@@                                      118
               8 |@@@@@                                    182
              16 |@@@@@                                    211
              32 |@@@@@@@@@@@@@@@@@                        689
              64 |@                                        31
             128 |@                                        29
             256 |@@                                       99
             512 |@                                        24
            1024 |@@@                                      135
            2048 |                                         5
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         10
         2097152 |                                         0
         4194304 |@                                        20
         8388608 |                                         0

There is still a lot of work to do and while that goes on, the code has
to remain in the FreeBSD perforce server. It isn't ready to get merged
into CVS-current yet.

I have asked the perforce-admins to mirror the project out to CVS (via
cvsup10.freebsd.org), but I'm not sure what the hold-up there is.

I had hoped that one or two of the Google SoC students would contribute
to this, but I only received one proposal and that wasn't for anything
that would help get DTrace/FreeBSD completed.

There are things people can do to help. Some of them are build related;
some are build tool related; some are user-land DTrace specific; and the
rest are kernel related. Speak up if you are interested in working on
this!

--
John Birrell
```

This page intentionally left blank

E

USDT Example

This appendix was contributed by Alan Hargreaves.

Throughout this book, the suggested strategies for tracing user-land applications typically end with using the pid provider to trace application internals, should easier, stable providers not be available. Using the unstable pid provider can be extremely complex, can be extremely time-consuming, and can make for some brittle and difficult-to-maintain scripts. It can take days to figure out how to extract the desired information from the running internals of an application. Another option exists for using DTrace to observe and analyze application software; if the source code is available, you can insert your own User Statically Defined Tracing (USDT) provider into the source code to provide the custom probes and arguments that you desire. For them to be available in the production environment, the modified source code must be recompiled and the new binaries deployed. USDT gives us a way to build an application-specific provider with probe names and arguments that make semantic sense in the context of the application. For example, a USDT provider for a database could provide probes named `query-start` and `query-end`, with arguments containing the query string and client and database name, for queries to be examined and their time measured.

Some time ago, Brendan Gregg asked fellow DTrace expert (and Australian) Alan Hargreaves if he'd like to write a USDT-based Bourne shell provider, because one didn't exist at the time. Such a provider would facilitate tracing of Bourne shell scripts, providing probes and arguments for the entry and return from subroutines and other events of interest. Alan didn't have experience with the Bourne shell code; however, within 24 hours, he had not only studied it enough but had a working Bourne shell provider designed and implemented. We've asked Alan to contribute this appendix of how he designed and implemented the USDT Bourne shell provider. This demonstrates the other option that may be considered: If stable providers do not already exist and using the pid provider becomes too complex, then the source code (if available) can be edited, recompiled, and redeployed in production.

USDT Bourne Shell Provider

Integrating USDT probes into the Bourne shell provides an excellent example of using this DTrace facility and of how to approach instrumenting applications.

Compared to SDT

Although USDT probes use similar macros, they differ slightly from their kernel counterpart, SDT:

We don't provide the argument types in the probe.

We do need a `.d` file that declares all of the probes and their stability.

There is an extra step in the compile/link process.

Defining the Provider

All of the probes that we will place into the source code need to be declared in a `.d` file. This file also contains a number of *pragma* lines defining the stability levels of the provider, module, functions, and arguments. We will go into more depth on this when we discuss stability. For now, let's look at a simple probe declaration. We'll be using this small USDT provider throughout this section.

```
provider world {
        probe loop(int);
}

#pragma D attributes Evolving/Evolving/Common provider world provider
#pragma D attributes Private/Private/Common provider world module
#pragma D attributes Private/Private/Common provider world function
#pragma D attributes Evolving/Evolving/Common provider world name
#pragma D attributes Evolving/Evolving/Common provider world args
```

In this example, our provider is called *world*, and it provides one probe called `loop` that has a single integer argument. Note that we've declared the argument types where we declared the probe. We could save this in a file called `probes.d`.

Adding a USDT Probe to Source

Let's start with a simple example that uses the probe declaration we just used.

```
#include <stdio.h>
#include <unistd.h>

int
main(int ac, char **av) {
        int i;
        for (i = 0; i < 5; i++) {
                printf("Hello World\n");
                sleep(2);
        }
}
```

This program prints "Hello World" five times with a two-second pause between each. Let's say we wanted to monitor the loop variable before we did the `printf()`. There is no easy way to get that value using the pid provider. We could modify the code by adding the bold lines shown here:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sdt.h>

int
main(int ac, char **av) {
        int i;
        for (i = 0; i < 5; i++) {
                DTRACE_PROBE1(world, loop, i);
                printf("Hello World\n");
                sleep(2);
        }
}
```

The include file `sys/sdt.h` contains all the macros and definitions that we need to add USDT probes, including the one that we use: `DTRACE_PROBE1()`.

This describes a probe in the provider world, named *loop*, that will have a single argument. As we saw in the previous section, it is an integer. It is critical that the probe name, provider name, and argument type(s) in the probe match what we declared in the `.d` file.

I can see your fingers itching to try this, so we can compile it as follows:

```
solaris# cc -c helloworld.c
solaris# dtrace -G -s probes.d helloworld.o
solaris# cc -o helloworld -ldtrace probes.o helloworld.o
```

The `-G` option to the `dtrace(1M)` command tells it to generate ELF files containing the embedded DTrace program. The probes specified in the files listed in the `-s` options are saved into these objects to be linked into the binary. This also goes through looking for the probe points in other objects and replaces them with NOP instructions so that the default is that the probes are disabled. The file associated with the `-s` argument is treated as a D program containing the declaration of the probe points.

If we run this outside of the DTrace framework, it will do exactly as we expected and print "Hello World" five times with a two-second break between each.

Let's take a look at it with DTrace:

```
solaris# dtrace -q -c ./helloworld -n '
world$target:::loop {
        printf("%s:%s loop = %d\n", probemod, probefunc, arg0);
}'
helloworld:main loop = 0
Hello World
helloworld:main loop = 1
Hello World
helloworld:main loop = 2
Hello World
helloworld:main loop = 3
Hello World
helloworld:main loop = 4
```

So, in this small example we have managed to make a variable visible at a point that would otherwise have been difficult to do, probably requiring debug statements and special builds. The only overhead we have is a few NOP instructions where the probe would be.

This is fine for if we only want to make a simple variable visible at a particular point in the code path. Quite often you will want to do a little bit more calculation in what you are making visible. If we only use the `DTRACE_PROBEn()` macros as we have here, that overhead will be added even when the probes are disabled. This gets just a little trickier because we need to run another `dtrace(1M)` command to generate an include file:[1]

```
solaris# dtrace -h -s probes.d
```

1. *http://dtrace.org/blogs/ahl/2006/05/08/user-land-tracing-gets-better-and-better/*

This will create probes.h. Inside probes.h we get some more useful macro definitions. Among these, we get macros of the form PROVIDER_PROBENAME_ENABLED(). If we want to verify that the probe is enabled before we do anything, we must modify helloworld.c like this:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sdt.h>
#include "probes.h"

Int
main(int ac, char **av) {
        int i;
        for (i = 0; i < 5; i++) {
                if (WORLD_LOOP_ENABLED()) {
                        /* Lots of stuff that takes time */
                        DTRACE_PROBE1(world, loop, i);
                }
                printf("Hello World\n");
                sleep(2);
        }
}
```

Now if we create the header file for this, we get an added bonus. We get a much cleaner-looking macro for our probe. We also get macros of the form PROVIDER_PROBENAME(), so we can replace the DTRACE_PROBE1() line with simply the following:

```
                        WORLD_LOOP(i);
```

which makes for substantially cleaner-looking code, especially if we are doing a lot of probes.

Stability

DTrace provides two types of stability attributes for entities such as built-in variables, functions, and probes: a *stability level* and an architectural *dependency class*. The stability level assists you in making risk assessments when developing scripts and tools based on DTrace by indicating how likely an interface or DTrace entity is to change in a future release or patch. The dependency class tells you whether an interface is common to all Solaris platforms and processors or whether the interface is associated with a particular architecture such as SPARC processors only. The two types of attributes used to describe interfaces can vary independently.

When declaring stabilities of a USDT provider, we need to document this for the full probe specification, including the arguments.

- The provider
- The modules
- The function
- The probes
- The type and number of arguments in each probe

DTrace describes interfaces using a triplet of attributes consisting of two stability levels and a dependency class. By convention, the interface attributes are written in the following order, separated by slashes:

```
name-stability / data-stability / dependency-class
```

The *name stability* of an interface describes the stability level associated with its name as it appears in your D program or on the `dtrace(1m)` command line.

The *data stability* of an interface is distinct from the stability associated with the interface name. This stability level describes the commitment to maintaining the data formats used by the interface and any associated data semantics. For example, the `pid` D variable (not the pid provider) is a stable interface: Process IDs are a stable concept in Solaris, and it is guaranteed that the `pid` variable will be of type `pid_t` with the semantic that it is set to the process ID corresponding to the thread that fired a given probe in accordance with the rules for stable interfaces.

The *dependency class* of an interface is distinct from its name and data stability and describes whether the interface is specific to the current operating platform or microprocessor.

In the *probes.d* file previously, we made the following definitions:

```
#pragma D attributes Evolving/Evolving/Common provider world provider
#pragma D attributes Private/Private/Common provider world module
#pragma D attributes Private/Private/Common provider world function
#pragma D attributes Evolving/Evolving/Common provider world name
#pragma D attributes Evolving/Evolving/Common provider world args
```

This says that the name and data stability of all but the module and function part of the probes is evolving. This means the interface might eventually become standard or stable but is still in transition. All efforts would be made to avoid incompatible change, but if any were required, they could occur only in a minor or major release. The module and function are marked private. This means that they are subject to change and simply cannot be relied upon for stability for this probe. The third part of the triplet means that the probes are common to all architectures.

The "Stability" chapter of the DTrace Guide discusses stability in great depth.

Case Study: Implementing a Bourne Shell Provider

Before you start, it is useful to have a good idea of exactly what probes you are interested in providing. In the case of the shell provider, we decided on the probes listed in Table E-1.

Table E-1 Probes

Probe	Description
`builtin-entry`	Fires on entry to a shell built-in command
`builtin-return`	Fires on return from a shell built-in command
`command-entry`	Fires when the shell execs an external command
`command-return`	Fires on return from an external command
`function-entry`	Fires on entry into a shell function
`function-return`	Fires on return from a shell function
`line`	Fires before commands on a particular line of code are executed
`subshell-entry`	Fires when the shell forks a subshell
`subshell-return`	Fires on return from a forked subshell
`script-start`	Fires before any commands in a script are exexuted
`script-done`	Fires on script exit
`variable-set`	Fires on assignment to a variable
`variable-unset`	Fires when a variable is unset

In addition, we need to think about the probe arguments (Table E-2).

Table E-2 Probe Arguments

Type	Argument	Description
builtin-entry, command-entry, function-entry		
`char *`	`args[0]`	Script name
`char *`	`args[1]`	Built-in/command/function name
`int`	`args[2]`	Line number
`int`	`args[3]`	# arguments
`char **`	`args[4]`	Pointer to argument list
builtin-return, command-return, function-return		
`char *`	`args[0]`	Script name
`char *`	`args[1]`	Built-in/command/function name
`int`	`args[2]`	Return value
subshell-entry		
`char *`	`args[0]`	Script name
`pid_t`	`args[1]`	Forked process ID
subshell-return		
`char *`	`args[0]`	Script name
`int`	`args[1]`	Return value
line		
`char *`	`args[0]`	Script name
`int`	`args[1]`	Line number
script-start		
`char *`	`args[0]`	Script name
script-done		
`char *`	`args[0]`	Script name
`int`	`args[1]`	Exit value
variable-set		
`char *`	`args[0]`	Script name
`char *`	`args[1]`	Variable name
`char *`	`args[2]`	Value
variable-unset		
`char *`	`args[0]`	Script name
`char *`	`args[1]`	Variable name

You will notice that we've tried to get as much consistency as possible: `args[0]` is *always* the script name. Within similar probes, we've also kept the arguments consistent. This makes it simpler to write scripts using these probes.

This makes for a `probes.d` that looks like this:

```
provider sh {
      probe function__entry(char *, char *, int, int, char **);
      probe function__return(char *, char *, int);
      probe builtin__entry(char *, char *, int, int, char **);
      probe builtin__return(char *, char *, int);
      probe script__start(char *);
      probe script__done(char *, int);
      probe command__entry(char *, char *, int, int, char **);
      probe command__return(char *, char *, int);
      probe subshell__entry(char *, pid_t, int);
      probe subshell__return(char *, int);
      probe line(char *, int);
      probe varible__set(char *, char *, char *);
      probe variable__unset(char *, char *);
};

#pragma D attributes Unstable/Unstable/Common provider sh provider
#pragma D attributes Private/Private/Unknown provider sh module
#pragma D attributes Private/Private/Unknown provider sh function
#pragma D attributes Unstable/Unstable/Common provider sh name
#pragma D attributes Unstable/Unstable/Common provider sh args
```

The stability of this provider is currently marked in general as unstable because it is under development. The module and function parts are private to the shell code, so no reliance should be placed on them. The probes are common to all architectures. Once the provider has had some use, we will look at firming up the stabilities.

Where to Place the Probes

Working out exactly where in the source code to place the probes can be difficult if you are just looking at the source. One way to find exactly which functions you need to place the probes in is to leverage the pid provider to look at which functions get executed with a small script that you would expect to make each probe fire (one at a time). For example, to catch likely *builtin-entry* and *builtin-return* locations, we could use the following script (prime numbers are good as you are less likely to hit them by accident):

```
#!/usr/has/bin/sh

for i in 1 2 3 4 5 6 7 8 9 10 11 12 13
do    echo $i
done
```

Note that as I am running this on an Solaris Development box, the Bourne shell is /usr/has/bin/sh, not /bin/sh.

We would then use the pid provider to watch function calls:

```
solaris# dtrace -n 'pid$target:sh::entry { @[probefunc] = count(); }' -c ./builtin.sh
```

Among a lot of other information here we find this:

```
  echo   13
```

We can now redo the probe to look at the stack when we call b_echo().

```
solaris# dtrace -n 'pid$target:sh:echo:entry { ustack(10); exit(0); }' -c ./
builtin.sh
[...]
CPU     ID                    FUNCTION:NAME
  0  62526                       echo:entry
              sh`echo
              sh`builtin+0x2d4
              sh`execute+0x571
              sh`execute+0x28a
              sh`exfile+0x195
              sh`main+0x518
              sh`_start+0x7d
```

After a bit of looking at the code at this point, it looks like execute() is probably going to be a good function to place the built-in probes. In the Bourne shell, this function is inside xec.c.

Within the function we find a switch() statement in which we can actually place a couple of other entry/return probes as well as the built-in entry and built-in return. The code we end up with is as follows:

```
else if (comtype == BUILTIN) {
        SH_BUILTIN_ENTRY(cmdadr, *com, t->line, argn, c);
        builtin(hashdata(cmdhash), argn, com, t);
        SH_BUILTIN_RETURN(cmdadr, *com, exitval);
        freejobs();
        break;
} else ...
```

The variable-set probe is also worth showing here as it needs to use an _ENABLED macro.

```
                if (SH_VARIABLE_SET_ENABLED()) {
                        char *value = (char *)argscan;
                        value++;
                        SH_VARIABLE_SET((char *)cmdadr, (char *)argi,
                            value);
                }
```

The value that we need to place into `args[1]` is one byte beyond where `argscan` is pointing. Now we don't want to do an increment in the macro for three reasons.

- We can't be sure how many times the macro instantiates the argument; we may end up incrementing by more than one.
- If we increment `argscan`, then we need to decrement it again for the following code to use it. The probe would then have an impact when disabled (an increment and decrement of a variable).
- If DTrace is undefined on the target system, we may even have zero instantiations, making a subsequent decrement (to put the value back where it should be) incorrect.

The solution was to place it in the previous clause and increment a stack variable *only* if the probe is enabled.

We also need to ensure that the files we added probes to include `probes.h` that we generate from `probes.d`. We also need to make the following additions and modifications (bold) to the Makefile to properly generate the code:

```
OBJS=   args.o blok.o cmd.o defs.o error.o fault.o hash.o hashserv.o \
        io.o msg.o print.o stak.o string.o word.o xec.o \
        ctype.o echo.o expand.o func.o macro.o pwd.o setbrk.o test.o \
        bltin.o jobs.o ulimit.o sh_policy.o main.o name.o service.o \
        probes.o

all: probes.h $(ROOTFS_PROG)
      dtrace -h -s probes.d

probes.o: error.o main.o xec.o name.o probes.d
      dtrace -32 -G -s probes.d error.o main.o xec.o name.o

clean:
      $(RM) probes.h $(OBJS)
```

It's also worth noting that there was actually a little more to this provider, because the way the shell was written, it did not keep track of line numbers beyond parsing. The *line* structure element and some support for it also needed to be added but is beyond the scope of this appendix.

This page intentionally left blank

F

DTrace Error Messages

DTrace will print error or warning messages to STDERR as a result of specific events. This appendix describes the more common messages encountered and provides suggestions for avoiding them.

Many of these are from the "DTrace Tips, Tricks and Gotchas" presentation[1] by Bryan Cantrill, Mike Shapiro, and Adam Leventhal.

Privileges

Message

```
macosx# dtrace -l
dtrace: failed to initialize dtrace: DTrace requires additional privileges
macosx#
```

Meaning

The user does not have the necessary permissions to run DTrace.

1. *http://dtrace.org/blogs/bmc/2005/02/28/dtrace-tips-tricks-and-gotchas/*

Suggestions

On Mac OS X and FreeBSD, the only solution is to run as the root user or use `sudo(8)`.

On Solaris systems, required privileges to run DTrace can be assigned using process privileges (also see Chapter 11, Security):

> `dtrace_user`: Allows the use of profile, syscall and fasttrap providers, on processes that the user owns
>
> `dtrace_proc`: Allows the use of the pid provider on processes that the user owns
>
> `dtrace_kernel`: Allows most providers to probe everything, in read only mode

Privileges can be added to a process (such as a user's shell) temporarily by using the `ppriv(1)` command. For example, to add dtrace_user to PID 1851, use this:

```
solaris# ppriv -s A+dtrace_user 1851
solaris# usermod -K defaultpriv=basic,dtrace_user brendan
```

Drops

Message

```
dtrace: 978 drops on CPU 0
```

Meaning

DTrace ran out of available principal buffer space for recording output data. This occurs when the switch buffer policy is in use (which is the default) and more data is output than there is space available in the active principal buffer. In practice, this usually happens when outputting thousands of events (and many pages of output) per second.

Suggestions

The size of the principal buffer can be increased by adjusting the `bufsize` tunable variable per consumer, which may eliminate drops. For example, to increase it from the default of 4MB per CPU to 8MB per CPU, you can use `-b 8m` or `-x bufsize=8m` on the command line, or you can use `#pragma D option bufsize=8m` in D scripts (see Appendix A).

Also, the `switchrate` tunable can be increased from the default of 1 Hertz to 10 Hertz using `-x switchrate=10hz` on the command line or `#pragma D option switchrate=10hz` in a D script, which can also reduce buffer drops because it is drained more quickly by the DTrace consumer.

Finally, it may be possible to modify the script to record less data, such as by using predicates to filter out uninteresting events or by using aggregations to summarize data instead of printing out everything. Either of these will relieve the pressure on the principal buffer, reducing drops.

Since this is a common error message, it is also described in Chapter 14, Tips and Tricks.

Aggregation Drops

Message

```
dtrace: 11 aggregation drops on CPU 0
```

Meaning

DTrace ran out of available aggregation buffer space for recording aggregation data.

Suggestions

The aggregation buffer size can be increased by setting the `aggsize` tuneable variable. To set the size to 8MB, either use `-x aggsize=8m` on the command line or use `#pragma D option aggsize=8m` in D scripts (see Appendix A).

Also, the rate of consumption of aggregation data can be tuned by increasing `aggrate` from the default of 1 Hertz to 10 Hertz using `-x aggrate=10hz` on the command line or `#pragma D option aggrate=10hz` in a D script.

Dynamic Variable Drops

Message

```
dtrace: 103 dynamic variable drops
dtrace: 73 dynamic variable drops with non-empty dirty list
```

Meaning

Space for dynamic variables (thread-local variables, associative array variables) has been depleted.

Suggestions

Ensure your D programs are clearing unused dynamic variables (for example, `self->myvar = 0;`).

The space for storage of dynamic variables can be increased from the default of 1MB (`dynvarsize`) to 2MB per consumer using `-x dynvarsize=2m` or in D scripts with `#pragma D option dynvarsize=2m` (see Appendix A).

If the message includes `non-empty dirty list`, the `cleanrate` variable can be increased from the default of 101Hz per-consumer using `-x cleanrate=333hz` on the command line or using `#pragma D option cleanrate=333hz` in a D script.

Note dynamic variable drops must be eliminated for correct results.

Invalid Address

Message

Invalid address (0x...) in action. For example:

```
# dtrace -n 'syscall::open:entry { trace(stringof(arg0)); }'
[...]
dtrace: error on enabled probe ID 1 (ID 6329: syscall::open:entry):
invalid address (0xd27fbf38) in action #1
```

Meaning

This error is caused when DTrace attempts to dereference a memory address that isn't mapped. In the previous example, the `arg0` variable for the `open(2)` system call refers to a user-land address; however, DTrace executes in the kernel address space.

Suggestions

Depending on the specific address being deferenced, the use of `copyin()` or `copyinstr()` may be required to copy a user-land address into the kernel before DTrace can dereference it. And so, this example may be fixed by changing the function from `stringof()` to `copyinstr()` in the `trace()` statement.

When derefencing addresses in an entry probe, it is also a good idea to move the dereference to the corresponding return probe if possible (which may involve saving the address as a thread-local variable on entry, for reference on return). On entry, the address may be valid, but the actual memory page(s) have not been faulted in yet. If that is the case, a similar error message is seen—despite using `copyinstr()`.

Maximum Program Size

Message

```
dtrace: failed to enable './biggie.d': DIF
program exceeds maximum program size
```

Meaning

The D program (`biggie.d`) comprised a very large number of enablings, and/or actions, exceeding DTrace's ability to execute the program because of size requirements for internal DTrace objects.

Suggestions

Edit the program, reducing the number of probes and/or actions. Alternative, consider increasing the `dtrace_dof_maxsize` variable in `/etc/system` (Solaris). The default value is 256KB.

Not Enough Space

Message

```
# dtrace -ln 'pid$target:::' -p `pgrep mozilla-bin`
dtrace: invalid probe specifier pid$target:::: failed to create probe in process 7424:
 Not enough space
```

Meaning

The limit on the number of pid provider probes that can be created (250,000) has been reached. This can happen when trying to instrument very large process using the pid provider.

Suggestions

Often, this can be the result of an unintended invocation by the user; `pid$target:::`, which will attempt to insert a probe at every instruction in the target process, was actually intended to be `pid$target:::entry` or `pid$target:libc::entry`, and so on. Note there is potentially a huge difference in the number of probes required for `pid$target:::` vs. `pid$target:::entry`. The user should modify the DTrace to require fewer probes for the target process.

If instrumenting large processes is required, consider increasing the pid probe limit by editing the `/kernel/drv/fasttrap.conf` file (in Solaris) and changing the `fasttrap-max-probes` value from 250000 to something larger. After editing, you will need to run `update_drv fasttrap` or reboot.

G

DTrace Cheat Sheet

Synopsis

```
dtrace -n 'probe /predicate/ { action; }'
```

Finding Probes

1. DTrace Guide: Currently at http://wikis.sun.com/display/DTrace/Documentation
2. Keyword search: `dtrace -l | grep foo`
3. Frequency count:
```
dtrace -n 'fbt:::entry { @[probefunc] = count(); }' \
-c 'ping host'
```
4. DTraceToolkit: `grep foo /opt/DTT/Bin/*`

Finding Probe Arguments

`syscall:::`	`man syscallname`
`fbt:::`	Kernel source, or `mdb -k` and `::nm -f ctype` (Solaris)
everything else	DTrace Guide

Probes

`BEGIN`	D program start
`END`	D program end
`syscall::read*:entry`	process reads
`syscall::write*:entry`	process writes
`syscall::open*:entry`	file open
`proc:::exec-success`	process create
`io:::start,io:::done`	disk or NFS I/O request, completion
`lockstat:::adaptive-block`	blocked thread acquired kernel mutex
`sysinfo:::xcalls`	CPU cross calls
`sched:::off-cpu`	thread leaves CPU
`fbt:::entry`	entire kernel - all function entry probes
`javascript*:::`	JavaScript provider probes
`perl*:::`	Perl provider probes
`profile:::tick-1sec`	run once per sec, one CPU only
`profile:::profile-123`	sample at 123 Hertz

Vars

`execname`	on-CPU process name	`probemod`	module name
`pid, tid`	on-CPU PID, Thread ID	`probefunc`	function name
`cpu`	CPU ID	`probename`	probe name
`timestamp`	time, nanoseconds	`self->foo`	thread-local
`vtimestamp`	time thread was on-CPU, ns	`this->foo`	clause-local
`arg0..N`	probe args (uint64)	`$1..$N`	CLI args, int
`args[0]..[N]`	probe args (typed)	`$$1..$$N`	CLI args, str
`curthread`	pointer to current thread	`$target`	Set via `-p` or `-c`
`curpsinfo`	procfs style process information	`zonename`	zonename

Actions

`@agg[key1, key2] = count()`	frequency count
`@agg[key1, key2] = sum(var)`	sum variable
`@agg[key1, key2] = quantize(var)`	power of 2 quantize variable
`printf("format", var0..varN)`	print vars; use `printa()` for aggregations
`stack(num), ustack(num)`	print num lines of kernel, user stack
`func(pc), ufunc(pc)`	return kernel/user function name from program counter
`clear(@)`	clear an aggregation
`trunc(@, 5)`	truncate an aggregation to top 5 entries
`stringof(ptr)`	string from kernel address
`copyinstr(ptr)`	string from user-land address
`exit(0)`	exit `dtrace(1M)`

Switches

`-n`	trace this probe description
`-l`	list probes instead of tracing them
`-q`	quiet - don't print default output
`-s file`	invoke script file; or at top of script: `#!/usr/sbin/dtrace -s`
`-w`	allow destructive actions
`-p PID`	allow `pid:::` provider to trace this PID; the PID is available as `$target`
`-c 'command'`	have `dtrace(1M)` invoke this command
`-o file`	append output to file
`-x options`	set various DTrace options (`switchrate`, `bufsize`, ...)

Pragmas

`#pragma D option quiet`	same as `-q`, quiet output
`#pragma D option destructive`	same as `-w`, allow destructive actions
`#pragma D option switchrate=10hz`	print at 10 Hertz (instead of 1 Hertz)
`#pragma D option bufsize=16m`	set per-CPU buffer size (default 4MB)
`#pragma D option defaultargs`	`$1` is 0, `$$1` is "", etc...

One-Liners

```
dtrace -n 'proc:::exec-success  { trace(curpsinfo->pr_psargs); }'
dtrace -n 'syscall:::entry      { @num[execname] = count(); }'
dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'
dtrace -n 'io:::start           { @size = quantize(args[0]->b_bcount); }'
dtrace -n 'fbt:::entry          { @calls[probemod] = count(); }'
dtrace -n 'sysinfo:::xcalls     { @num[execname] = count(); }'
dtrace -n 'profile-1001         { @[stack()] = count() }'
dtrace -n 'profile-101 /pid == $target/ { @[ustack()] = count() }' -p PID
dtrace -n 'syscall:::entry      { @num[probefunc] = count(); }'
dtrace -n 'syscall::read*:entry { @[fds[arg0].fi_pathname] = count(); }'
dtrace -n 'vminfo:::as_fault    { @mem[execname] = sum(arg0); }'
dtrace -n 'sched:::off-cpu /pid == $target/ { @[stack()] = count(); }' -p PID
dtrace -n 'pid$target:libfoo::entry { @[probefunc] = count(); }' -p PID
```

Bibliography

Suggested Reading

This practitioner-oriented list comprises the core texts necessary for an understanding of the areas covered in this book. It may be read in conjunction with the bibliographies contained in the appropriate operating systems or languages texts. For the Solaris operating system, the text is *Solaris Internals* (McDougall et al, 2006), referenced here, which lists a broader collection of academic white papers and textbooks.

Aho, A.V., B.W. Kernighan, and P.J. Weinberger. 1988. *The AWK Programming Language*. Reading, MA: Addison-Wesley. This book is useful as a guide to postprocessing D output in awk, the principles of which carry over into Perl.

Cantrill, B., M. Shapiro, and A. Leventhal. 2005. *DTrace Tips, Tricks, and Gotchas* (presentation). Written by team DTrace, this presentation can be considered an extension to Chapter 14, Tips and Tricks, because it continues to cover some more advanced topics.

Dougherty, D. 1997. *sed and awk*, 2nd ed. Sebastopol, CA: O'Reilly. This is a comprehensive guide to postprocessing languages for DTrace output—not as steep a climb as Aho et al. (1988).

Flanagan, D. 2006. *JavaScript: The Definitive Guide*, 5th ed. Sebastopol, CA: O'Reilly.

Flanagan, D., and Y. Matsumoto. 2008. *The Ruby Programming Language*. Sebastopol, CA: O'Reilly.

Gove, D. 2007. *Solaris Application Programming*. Upper Saddle River, NJ: Prentice Hall. This is a wide-ranging look at application performance with usefully distilled material on the compiler and CPU counters.

Horstmann, C.S., and G. Cornell. 2008. *Core Java, Volume I: Fundamentals*, 8th ed., and *Core Java, Volume 2: Advanced Features*, 8th ed. There are many books on many aspects of Java. These are two of the most comprehensive.

Kernighan, B.W., and D.M. Ritchie. 1988. *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall. The C programming language is described by the authors, albeit in terms too terse for some who may prefer gentler introductions.

Lerdorf, R., K. Tatroe, and P. MacIntyre. 2006. *Programming PHP*, 2nd ed. Sebastopol, CA: O'Reilly.

Lutz, M. 2009. *Learning Python*, 4th ed. Sebastopol, CA: O'Reilly.

Maguire, A. 2010. *The DTrace TCP Provider: An Introduction for System Administrators* (white paper).

McDougall, R., and J. Mauro. 2006. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed. Upper Saddle River, NJ: Prentice Hall. This is the definitive work on the Solaris kernel. It's invaluable if you are using DTrace on that platform.

McDougall, R., J. Mauro, and B. Gregg. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Upper Saddle River, NJ: Prentice-Hall. Accompanying work to McDougall and Mauro (2006), this book rounds out the use of other Solaris observability tools and techniques.

Neville-Neil, G.V., and M.K. McKusick. 2004. *The Design and Implementation of the FreeBSD Operating System*. Boston, MA: Addison-Wesley. This covers the internals of the BSD OS for those using that platform.

Ousterhout, J.K. 2009. *Tcl and the Tk Toolkit*, 2nd ed. Boston, MA: Addison-Wesley.

Singh, A. 2006. *Mac OS X Internals: A Systems Approach*. Boston, MA: Addison-Wesley. This is a weighty tome on the internals of Mac OS X; it's required reading for those preferring that platform.

Stevens, W.R. 1993. *TCP/IP Illustrated: Vol 1: The Protocols.* Reading, MA: Addison-Wesley. The two chapters on networking of this book presume a knowledge of networking fundamentals admirably addressed by Stevens and the second and third volumes of this series on the networking implementation of BSD and HTTP protocol.

Stevens W.R. 1998. *Unix Network Programming: Interprocess Communications v. 2.* Reading, MA: Addison-Wesley. This book details the interfaces for networking within the single system.

Stevens, W.R., B. Fenner, and A.M. Rudoff. 2003. *Unix Network Programming: Networking APIs, Sockets & XTI*. Boston, MA: Addison-Wesley. The programmatic interfaces to the networking stack are covered in detail.

Stevens, W.R., and S.A. Rago. 2005. *Advanced Programming in the Unix Environment*. Boston, MA: Addison-Wesley. The second edition of the classic work lays the foundation for a clear understanding of both this book and other works in this bibliography.

Stroustrup, B. 2000. *The C++ Programming Language.* Boston, MA: Addison-Wesley. This is a definitive text for those DTrace-ing applications written in this language.

Sun Engineers, 2009. *Solaris 10 Security Essentials*. Upper Saddle River, NJ: Prentice Hall. This is background reading to Chapter 11, Security.

Teer, R. 2004. *Solaris Systems Programming.* Upper Saddle River, NJ: Prentice Hall. This is a treatment of the Solaris kernel/user interfaces in the style of Stevens' *Advanced Programming in the Unix Environment.* A desirable adjunct to McDougall and Mauro (2006).

Wall L., T. Christiensen, and J. Orwant. 2000. *Programming Perl*, 3rd ed. Sebastopol, CA: O'Reilly.

Vendor Manuals

FreeBSD

FreeBSD Architecture Handbook: *www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html*

FreeBSD Developers' Handbook: *www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/*

Mac OS X

I/O Kit Fundamentals: *http://developer.apple.com/mac/library/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/*

Kernel Programming Guide: *http://developer.apple.com/mac/library/documentation/Darwin/Conceptual/KernelProgramming/About/About.html*

Solaris

All of the following locations are expected to change from Sun.com to Oracle.com, although the new locations are not yet known.

Device Driver Tutorial: *http://docs.sun.com/doc/817-5789*

DTrace User Guide: *http://docs.sun.com/doc/819-5488*

DTrace Documentation Wiki: *http://wikis.sun.com/display/DTrace/Documentation*

Linker and Libraries Guide: *http://docs.sun.com/doc/817-1984*

Multithreaded Programming Guide: *http://docs.sun.com/doc/816-5137*

Programming Interfaces Guide: *http://docs.sun.com/doc/817-4415*

Solaris Dynamic Tracing Guide: *http://docs.sun.com/doc/817-6223*

Solaris Modular Debugger Guide: *http://docs.sun.com/doc/816-5041*

SPARC Assembly Language Reference Manual: *http://docs.sun.com/doc/816-1681*

Writing Device Drivers: *http://docs.sun.com/doc/816-4854*

x86 Assembly Language Reference Manual: *http://docs.sun.com/doc/817-5477*

Glossary

action In DTrace, the term *action* refers to the action statements taken when a probe fires, which are defined in an action clause. See *clause*.

adaptive mutex A mutex (mutual exclusion) lock type. When requested yet held by another thread, it will either spin if that other thread is currently executing on another CPU or block if it isn't.

aggregating function A D built-in function that operates on aggregations. These include population functions, such as `count()`, `avg()`, `sum()`, and `quantize()`; processing functions such as `normalize()` and `trunc()`; and `printa()` for printing. (See the "Aggregations" section in Chapter 2.)

aggregation A special D variable type used to summarize data. They are prefixed with an at (@) sign, are populated by aggregating functions on a per-CPU basis, and are combined only when printed out; this minimizes the overhead on multi CPU systems. See *aggregating function*.

aggregation buffer A per-CPU buffer for aggregation data. This can be tuned by the `aggsize` tunable.

Analytic A graphical interface for performance analysis, which uses DTrace to provide much of its data. It is shipped as part of the Oracle ZFS Storage Systems.

anchored probes Probes that instrument a specific location in code, such as the fbt provider entry and return probes, which instrument the entry and return of kernel functions. See *unanchored probes*.

API Application Programming Interface.

array A variable type that consists of a set of values, referenced by an integer index. A string is an array of characters.

associative array A collection of values that are each assigned and retrieved using a unique key. These differ from ordinary arrays in that the key (index) can be something other than a consecutive series of integers. They are often used to store data that may be retrieved by different threads, keyed on some global identifier such as a buffer address.

buffer A region of memory used for temporary data. With DTrace, this is sometimes used to refer to the DTrace principal buffer (see *principal buffer*).

bufpolicy DTrace tunable parameter to define the principal buffer behavior. Values can be `switch` (default), `fill`, or `ring`. See *principal buffer.*

C The C programming language.

C++ The C++ programming language.

cast See *type cast*.

Chime A graphical tool for executing and displaying DTrace data. Chime uses the Java DTrace API.

CIFS Common Internet File System. A file system and device protocol commonly used by Microsoft Windows (sometimes referred to as *Server Message Block* [SMB]).

clause A series of one or more action statements that are executed when a probe fires, grouped in braces, as in `{ ... }`.

clause local A D variable type intended for use within an action clause, `{ ... }`, for temporary variables.

command A program executed at the shell.

comment characters The characters `/* ... */`, which indicate that text is not to be executed. The text is usually provided to help other programmers understand the code, by including description of the code.

commented out Taking a body of code and encapsulating it in the comment characters `/* ... */` so that it is removed from the execution of the program.

consumer End user. Consumers of the DTrace framework are the user-level commands including `dtrace(1M)` and `lockstat(1M)`, which consume DTrace via the libdtrace library.

core A execution pipeline on a CPU.

CPU Central Processing Unit. The microprocessor hardware of the computer.

CR Change Request. This is the term Sun Microsystems uses for filed bugs, which are requesting a change in a product. CR is followed by the numeric bug ID and a text synopsis, such as "CR 6558517: need DTrace versions of IP address to string functions, like inet_ntop()."

cross calls A CPU cross call is when one CPU sends a request to others on a multi-CPU system. These can be for systemwide events such as cache coherency.

CTF Compact C Type Format. Debugging information that is built in to the executable to describe C types (structures, typedefs, and so on) and function prototypes.

D D is the programming language supported by DTrace and processed by `dtrace(1M)` and `libdtrace(3LIB)`. It was inspired by the C programming language.

D language See *D*.

D program A program written in the D programming language. It may be executed at the command line or saved to a file and executed as a D script.

D script A D program saved as a file and executed with either `dtrace -s filename` or by making the file executable and adding an interpreter line such as `#!/usr/sbin/dtrace -s`. By convention, it has a `.d` extension.

DIF DTrace Intermediate Format. An encoding of RISC-like instructions used to represent predicates and actions bound to DTrace probes.

DIFO DIF Object. Representation of a D expression for evaluation.

dispatcher queue Also known as a *run queue*. A queue of runnable threads.

DNS Domain Name Service.

DOF DTrace Object Format. An encoding of a DTrace program.

drops Data was dropped, so the output of DTrace is incomplete. This happens when using the switch buffer policy (which is the default) and the active principal buffer has filled because of a high rate of trace data. See "drops and dynvardrops" in Chapter 14.

DTrace The term *DTrace* alone refers to either the technology or the kernel implementation.

dtrace(1M) Command-line program for executing D programs either as one-liners or as scripts.

dtrace(7d) DTrace kernel implementation (the term is the Solaris man page name).

DTraceToolkit This is the name of a freely downloadable software package containing a large collection of D scripts and one-liners with supporting documentation.

dynamic This term is often used with DTrace to refer to dynamic probes, which is included in the name of the technology (Dynamic Tracing). Dynamic probes are automatically generated by DTrace, such as the instrumentation of currently running software to provide function entry and return probes. Providers such as fbt and pid provide dynamic probes. Depending on the target software, tens or hundreds of thousands of dynamic probes may be available. See *static*.

dynvardrops The dynamic variable buffer has filled, and assignments are dropped. The output from DTrace may no longer be accurate with the presence of `dynvardrops`. See *switch buffer*, and also see "drops and dynvardrops" in Chapter 14.

enabling A group of enabled probes and their associated predicates and actions.

errno The error value returned by the last system call made by the current thread. These are defined in `errno.h` and `intro(2)`. For example, `errno 2` is usually `ENOENT` "No such file or directory."

execname Process name that was passed to `exec()`.

fault A possible failure mode of hardware and software. An expected error event.

fbt provider Function Boundary Tracing provider. Dynamically provides probes for the entry and return of kernel functions.

FC Fibre Channel. A block storage protocol.

fds[] array A built-in D array currently available on Solaris and Mac OS X that provides translation from integer file descriptors into `fileinfo_t`, which contains the path name, mount point, file system type, and more.

Fibre Channel See *FC*.

fill buffer A principal buffer policy that has a fixed size and fills once. When one is full, tracing stops and `dtrace(1M)` exits, processing all the CPU fill buffers. This is set using `bufpolicy=fill` (see *bufpolicy*).

FreeBSD A Unix-like operating system, based on BSD. FreeBSD is generally regarded as reliable, robust, and secure and has a number of security enhancements from the TrustedBSD project.

FTP File Transfer Protocol. A commonly used protocol to transfer files over the Internet.

GLDv3 Generic LAN Driver version 3. Provides an interface for local area network (LAN) device drivers on Solaris.

global See *scalar*.

global scalar variables See *scalar*.

global variables See *scalar*.

Hertz Cycles per second.

HFS+ Hierarchical File System Plus. A file system developed by Apple used in Mac OS X.

HTTP HyperText Transfer Protocol.

ICMP Internet Control Message Protocol. Used by `ping(1)` (ICMP echo request/reply).

IDE Integrated Drive Electronics. An obsolete interface standard for storage devices.
Integrated Development Environment. A GUI-based software application for developing software.

inline Refers to code that is included in another body of code, rather than referring to code that is saved in a separate file.

IP Internet Protocol. The term *ip* may also refer to the Solaris kernel ip module that implements network stack protocols, including IP and TCP.

iSCSI Internet Small Computer Systems Interface. An IP-based storage protocol.

Java The Java programming language.

JavaScript The JavaScript programming language.

KB Kilobytes.

kernel The master program on a system that runs in privileged mode to manage resources and user-level processes.

kernel-land Also known as *kernel mode* or *system mode*. A virtual memory–based operating system supports multiple execution modes that define the hardware and software context of the running software, including the addressable memory or address space. Kernel-land refers to executing in a privileged context, and the kernel address space, in which the kernel and device drivers execute. DTrace probes also fire in kernel-land, which is why user-land addresses including the path name pointer for the `open()` system call cannot be dereferenced directly; rather, a D function (`copyin()` or `copyinstr()`) must first be used to copy data from the user address space to the kernel. See *user-land*.

keys Used to refer to the identifier for key/value pairs in associative arrays.

latency The time for an event, such as the time for an I/O to complete. Latency is important for performance analysis, because it is often the most effective measure of a performance issue.

LDAP Lightweight Directory Access Protocol. A name service protocol.

libdtrace(3LIB) The C library interface used by DTrace consumers (for example, `dtrace(1M)` and `lockstat(1M)`) to access the kernel DTrace framework. `libdtrace(3LIB)` includes the D language compiler and facilities for enabling probes and consuming trace data. This library is currently a private interface and not for public consumption; it is subject to change at any time without notice.

lockstep This term is used in DTrace to refer to sampling at the same rate as another timed event, which could over-represent the event in the collected sample data.

Mac OS X A Unix-based operating system and graphical user environment developed by Apple Inc.

Mac OS X Instruments A graphical analyzer for Mac OS X that uses DTrace.

macro A method of generating variables in the D programming language. (See Chapter 2.) Macros are processed and replaced with literal text by the m4 preprocessor, part of the C compiler tool chain. See *m4(1)* and the Solaris Programming Utilities Guide (*http://docs.sun.com/app/docs/doc/801-6734/*).

malloc Memory allocate. This usually refers to the function performing allocation.

MB Megabytes.

memory This term is used to refer to system memory, which is usually implemented as DRAM.

MIB Message Information Base. These describe the data served via SNMP.

MMU Memory Management Unit. This is responsible for presenting memory to a CPU and for performing virtual to physical address translation.

mpt An SCSA-compliant nexus device driver.

mutex See *mutual exclusion lock*.

mutual exclusion lock Called *mutex* locks for short, these are software locks where only the thread that holds the lock can access the locked resource. This prevents simultaneous writing, which would otherwise result in data corruption on multi-CPU systems. Mutex locks are used throughout the kernel.

MySQL An open source relational database management system.

NetBeans DTrace GUI A graphical interface for DTrace currently available with the NetBeans IDE.

NFS Network File System. A protocol for accessing a file system over a network.

NFSv3 NFS version 3.

NFSv4 NFS version 4.

NIS Network Information Service. A name service protocol created by Sun Microsystems.

off-CPU A thread that is not currently running on a CPU and so is "off-CPU," because of having blocked on I/O or a lock, because of having yielded, or because it is waiting on a dispatcher queue.

on-CPU A thread that is currently running on a CPU.

onnv An abbreviation of ON Nevada, where ON is the consolidation of the Operating System and Networking components of Solaris. See *Solaris Nevada*.

OpenSolaris This refers to a development version of the Solaris operating system that was open to community contributions. The project has now been retired.

Oracle Solaris See *Solaris*.

Oracle Solaris Studio A software development platform for multiple languages, including C and C++. This includes DLight, a performance analyzer that is DTrace based.

OS Operating System. The collection of software including the kernel for managing resources and user-level processes.

OSs Operating Systems.

pagefault A system trap that occurs when a program references a memory location that is not currently part of its address space.

page-in/page-out Functions performed by an operating system (kernel) to move chunks of memory (pages) to and from external storage devices.

Perl The Perl programming language.

PHP The PHP programming language (originally, “Personal Home Page” tools).

PID Process Identifier. The operating system unique numeric identifier for processes.

pid provider Process ID provider. Dynamically provides probes for the entry, return, and instructions of user-level functions.

POSIX Portable Operating System Interface for Unix. A family of related standards by the IEEE to define a Unix API.

PostgreSQL An open source object-relational database management system.

pragma A compiler preprocessor directive.

predicate A D conditional statement, `/ ... /`, that evaluates either true or false. (See Chapter 2.)

principal buffer The main DTrace buffer that records the output of tracing actions including `trace()`, `printf()`, and `stack()`. It is per-CPU and 4MB by default, which can be tuned with the bufsize tunable (see Appendix A). Its behavior can be tuned with various buffer policies: see *switch buffer*, *fill buffer*, *ring buffer*.

probe A DTrace point of instrumentation, described by the four-tuple provider:module:function:name. Thousands of possible probes are available to DTrace, created either statically or dynamically. D programs enable probes and may take custom actions when they fire, such as printing data. See *static*, *dynamic*, *anchored probes*, and *unanchored probes*. (See Chapter 2.)

process An operating system abstraction of an executing user-level program. Each is identified by its PID (see *PID*) and may have one or more running threads (see *thread*).

profile The name of the profile provider, which can sample events at a given frequency. The term *profile* can also mean any technique to collect data that characterizes the performance of software.

provider A DTrace provider is a library of related probes and arguments. The provider name is specified as the first member of the probe name.

PSARC Platform Software Architecture Review Committee. A committee created at Sun Microsystems to review and approve most software developments, especially those that create or modify existing interfaces to users or other parts of the system, before integration.

Python The Python programming language.

reader/writer lock A mutual exclusion primitive used by threaded software to protect shared data.

RFC Request For Comments. This is a misleading acronym. In practice, an RFC is used by the Internet Engineering Task Force (IETF) as the document to define a protocol standard; in other words, RFC 793 defines the TCP protocol.

ring buffer A principal buffer policy that wraps when full, thereby only keeping recent events. A D program can then be written to exit on an event of interest, which will then process the ring buffer showing events that led up to that event. This is set using `bufpolicy=ring` (see *bufpolicy*).

Ruby The Ruby programming language.

sample In DTrace, the term *sample* is often used to refer to the time interval–based capturing of data. As such, only a sample of the data is captured and examined—rather than tracing every event. The profile provider samples at a specified rate. For a different technique of data collection, see *trace*.

SAS Serial Attached SCSI.

SATA Serial Advanced Technology Attachment. An interface standard for storage devices.

scalar A scalar variable. These are individual fixed-size data objects, such as integers and pointers (see Chapter 2). The term *global* is often used in addition as a reminder that it has global scope.

scalar global See *scalar.*

scalar variables See *scalar.*

SCSI Small Computer System Interface. An interface standard for storage devices.

SDT Statically Defined Tracing, kernel-based. This involves the placement of static DTrace instrumentation in kernel code by the kernel engineer, at locations to provide useful probes. SDT-based providers with a "stable" interface include io, proc, and sched; the sdt provider (*sdt* in lowercase, not to be confused with SDT) has an "unstable" interface. Also see *USDT.*

sdt provider A kernel DTrace provider for static probes, with a commitment level of "unstable." sdt probes are typically placed by kernel engineers as debug points and to prototype possible future stable providers.

self-> Prefix for thread-local variable. See *thread local*.

Shell A command-line interpreter and scripting language.

SMB Server Message Block protocol, also known as *CIFS* (see *CIFS*).

SNMP Simple Network Management Protocol.

snv See *Solaris Nevada*.

socket A software abstraction representing a network endpoint for communication.

Solaris A Unix operating system originally developed by Sun Microsystems. It is popular for enterprise use and is known for scalability, for reliability, and for introducing innovative features such as DTrace and ZFS. It has now been named Oracle Solaris after the acquisition of Sun by Oracle Corporation.

Solaris Nevada An Oracle, Inc., internal name for the current development version of Solaris, which is expected to be called Solaris 11 when released.

spin A software mechanism involving executing in a tight loop while trying to acquire a resource, typically a spin lock or an adaptive mutual exclusion (mutex) lock.

SSH Secure Shell. A encrypted remote shell protocol.

stable Used throughout this book to refer to the commitment level of a programming interface, usually the interface presented by a DTrace provider. A stable interface is one that should remain unchanged over time. D programs that use stable provider interfaces should work on future software versions without needing changes. See *unstable*.

stack Short for "stack trace."

stack backtrace See *stack trace*.

stack frame A data structure containing function state information, including pointers to the function, return address, and function arguments.

stack trace A call stack composed of multiple stack frames, showing the ancestry of executing functions. Reading a stack trace from bottom to top shows which functions have called which other functions and, from this, the path through code. This is also called a *stack back trace*, since reading the stack from top down begins with the most recent function and works backward to the least recent.

static Often used in DTrace to refer to statically defined probes, which are inserted into the source code by the programmer. Statically defined probes include those from the io and proc providers. DTrace will typically have dozens of statically defined probes available, depending on the available providers. See *dynamic*.

STDOUT The POSIX file descriptor name for normal command output.

subroutine A behavior implemented by the DTrace framework that can be performed at probe-firing time that modifies internal DTrace state but does not trace any data. Similar to actions, subroutines are requested using the D function call syntax.

switch buffer The default DTrace principal buffer policy that switches between active and inactive buffers, allowing the consumer (usually `dtrace(1M)`) to read the inactive buffer while the kernel continues to write to the active one. See *switchrate* and *drops*.

switchrate A DTrace tunable paramater that defines the rate of switching between active and inactive principal buffers, when the switch buffer policy is used. This is also the rate that the consumer reads from the switch buffers. The default is one second, which can introduce noticeable lag when watching `dtrace(1M)` output at the command line, and so for many scripts in this book, it is tuned to 10 Hertz. See *switch buffer*.

syscall System call. The interface for processes to request privileged actions from the kernel.

Tcl The Tcl programming language.

TCP Transmission Control Protocol. Originally defined in RFC 793.

this-> Prefix for clause-local variables. See *clause local*.

thread A software abstraction that represents a schedulable and executable component of a program. Typically a subset of a process.

thread local A D variable type that is associated with an individual thread, allowing data to be saved on a per-thread basis.

TLB Translation Lookaside Buffer. A cache for memory translation on virtual memory systems, used by the MMU (see *MMU*).

trace In DTrace, this term is used to refer to the inspection of every event. For example, the syscall provider can trace the entry point of every system call. For a different technique of data collection, see *sample*. This is also the name of the `trace()` built-in, which prints the argument given.

translator A collection of D assignment statements that convert implementation details of a particular instrumented subsystem into an object of struct type that forms an interface of greater stability than the input expression.

type cast Variables are of a type (integer, character, and so on). To cast a variable is to indicate to the compiler (or here, the Dtrace tool chain) that you want to treat a variable as a different type. This results in the size of the variable changing, which (seamlessly) affects pointer arithmetic.

uberblock Top or root block of the ZFS file system hierarchy.

UDP User Datagram Protocol. Originally defined in RFC 768.

unanchored probes Probes not associated with a specific location in code. Examples include the profile provider's profile and tick probes and the cpc provider probes. See *anchored probes*.

uncomment Remove the comment characters `/* ... */` from code.

unstable Used throughout this book to refer to the commitment level of a programming interface, usually the interface presented by a DTrace provider. An unstable interface is one that may change over time across different software versions; D programs written to use an unstable provider will need maintenance to match the provider as it changes. See *stable*.

USDT User-land Statically Defined Tracing. This involves the placement of static DTrace instrumentation in application code by the programmer, at locations to provide useful probes. USDT-based providers include plockstat, perl, python, ruby, javascript, hotspot, and X.

user-land Also known as *user-mode*. A virtual memory–based operating system supports multiple execution modes that define the hardware and software context of the running software, including the addressable memory or address space. user-land refers to the per-process address space for threads running in a nonprivileged mode, with access only to memory that is mapped to its address space. When a thread executes a system call, it causes a system trap that switches the mode to kernel-mode so the kernel can perform an operation, such as reading or writing a file, on behalf of the calling process. See *kernel-land*.

variable A named storage object. D variable types are summarized in Chapter 2.

workload The requests for a system or resource. For example, the workload on an NFS server can be described by the NFS protocol operations requested. Characteristics used to describe a workload typically include the number of clients, the type of requests (read, write, synchronous write), the rate and size of I/O, and whether the access pattern is generally sequential or random.

XDR External Data Representation. An encoding standard used by NFS.

ZFS A combined file system and volume manager created by Sun Microsystems.

Index

D

E

G

H

L

M

N

Q

R