Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels

Paul E. McKenney
B.S., Mechanical Engineering, Oregon State University, 1981
B.S., Computer Science, Oregon State University, 1981
M.S., Computer Science, Oregon State University, 1988

A dissertation submitted to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 2004

© Copyright 2004 by Paul E. McKenney
All Rights Reserved

The dissertation "Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels" by Paul E. McKenney has been examined and approved by the following Examination Committee:

Jonathan Walpole
Professor
Thesis Research Adviser

Andrew Black
Professor

Mark P. Jones
Associate Professor

Orran Krieger
IBM Research
Adjunct Professor
U. of Toronto
Carnegie-Mellon U.

Michael Scott
Professor
University of Rochester

Dedication

To my wife, Gloria Wyffels, and to our children, Melissa, Sarah, and Aaron.

Acknowledgements

I am indebted to both Jack Slingwine, co-inventor of RCU whose implementation within ptx/CLUSTERS was the first to ship to DYNIX/ptx customers, and to Dipankar Sarma, who did much of the hard work that lead to RCU's widespread use within the Linux 2.6 kernel, for extremely exciting and productive collaborations.

I am grateful to to Dave Stewart, Robin O'Neill, and Mike Paschal, who were willing to stake their respective products' release dates and reputations on RCU. Given that less than a year prior, RCU had been but a topic of discussion, this willingness bespeaks intestinal fortitude of mythic proportions.

I owe thanks to Stuart Friedberg, Doug Miller, Jan-Simon Pendry, Chandrasekhar Pularmarsetti, Peter Strazdins, and Dave Wolfe for their willingness to try out RCU, and to Ken Dove, to Brent Kingsbury, to Phil Krueger and his Base-OS Reading Group, and to my Bangalore students for many helpful discussions.

I am grateful to Wayne Cardoza, who very patiently explained the intricacies of the Alpha CPU's memory barriers, and similarly to Gary Oliver, who explained OS-3's use of the CDC 3300's interrupt system.

I am indebted to James Hennessy, Damian Osisek, and Joseph Seigh for discussions on the use of passive serialization in IBM's VM/XA product, to Luke Browning, Carl Burnett, Tom Mathews, and James Moody, for discussions on related topics in AIX, and to the K42 and Tornado groups, including Orran Krieger, Jonathan Appavoo, Marc Auslander, Dilma Da Silva, David Edelsohn, Michal Ostrowski, Bryan Rosenburg, Bob Wisniewski, and Jimi Xenidis, for excellent discussions and exchanges surrounding K42 generations, the combination of RCU and non-blocking synchronization, and alternative implementations of RCU infrastructure in Linux.

I am similarly indebted to a great number of Linux-community members. Dipankar Sarma implemented Linux's RCU infrastructure, and, together with Maneesh Soni, took on some of the less newbie-friendly portions of Linux. Andi Kleen, Rusty Russell, and Andrea Arcangeli each created alternative RCU-infrastructure implementations, using extremely clever and unexpected techniques. People participating in discussions, producing patches, trying out RCU, and otherwise keeping me honest include Al Dunsmuir, Al Viro, Alan Cox, Alexey Kuznetsov, Andrew Hutton, Andrew Morton, Andrew Tridgell, Andrey Panin, Anton Blanchard, Arnaldo Carvalho de Melo, Balbir Singh, Ben LaHaise, Benjamin Herrenschmidt, Bill Hartner, Bill Irwin, Brian Gerst, Christoph Hellwig, Cliff White, Corey Minyard, Daniel Phillips, Dave Hansen, Dave Jones, David Miller, Davide Libenzi, Don Marti, Ed Tomlinson, Erich Focht, Erik Anderson, Erlend Aasland, Fabian Frederick, George Anzinger, Gerrit Huizenga, Greg Kroah-Hartman, Hanna Linder, Hugh Blemings, Hugh Dickens, Ingo Molnar, Ivan Kokshaysky, J. W. Schultz, James Morris, Jamie Lokier, John Levon, Jonathan Corbet, Kai Henningsen, Karim Yaghmour, Keith Owens, Ken Rozendal, Kevin O'Connor, Luca Barbieri, Manfred Spraul, Marc-Christian Peterson, Mark Fasheh, Mark Hahn, Martin Bligh, Matt Dobson, Matt Mackall, Michael Baxter, Mikael Pettersson, Mingming Cao, Mitchell Blank Jr., Nikita Danilov, Oliver Neukum, Paul Mackerras, Pavel Machek, Peter Waechtler, Ravikiran Thirumalai, Richard Henderson, Rik Van Riel, Robert Love, Robert Olsson, Roman Zippel, Sam Ravnborg, Stephen Hemminger, Ted T'so, Thomas Schlichter, Trond Myklebust, Troy Wilson, Vamsi Krishna, Victor Yodaiken, Werner Almesberger, Zwane Mwaikambo, and Linus Torvalds. I learned at least as much about RCU in the past three years working with the Linux and K42 communities as I did in years prior.

I owe thanks to Bart Massey and Robert Bauer, who, much to my surprise, showed me that a formal definition of RCU semantics might well have practical applications.

I owe a debt of gratitude to my managers, Dale Goebel, Leslie Swanson, Rick Warren, Daniel Frye, Mark Dean, Chris Maher, Juergen Deicke, and Jai Menon, for their support of my work with RCU over more years than I care to count, and to my program committee, Jon Walpole, Andrew Black, Mark Jones, Orran Krieger, Michael Scott, for their inspiration and guidance.

This work was done with the aid of Macsyma, a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U.S. Department of Energy under grant ET-78-C-02-4687, and by the U.S. Air Force under grant F49620-79-C-020, between 1982 and 1992 by Symbolics, Inc. of Burlington Mass., and since 1992 by Macsyma, Inc. of Arlington, Mass. Macsyma is a registered trademark of Macsyma, Inc. May it rest in peace.

Other portions of this work were done with the aid of Maxima, a large open-source symbolic manipulation program.

SPEC™ and the benchmark name SPECweb™ are registered trademarks of the Standard Performance Evaluation Corporation. The benchmarking was done for research purposes only and may not be compared to published results on the SPECWeb site, due to the following deviations from the rules:

1. It was run on hardware that does not meet the SPEC availability-to-the-public criteria. The machine was an engineering sample.

2. `access_log` was not kept for full accounting. It was being written, but deleted every 200 seconds.

For the latest SPECweb99 benchmark results visit `http://www.spec.org`.

I owe thanks to OSDL for running many benchmarks used to evaluate RCU on Linux, and no dissertation is complete without an expression of thanks to Google.

I am grateful to Sequent Computer Systems, Inc. for providing the opportunity and hardware needed for the original development of RCU, as well as to IBM for continuing to provide such an environment, for their generous sharing of RCU with the Linux community, and for their wholehearted ongoing support of the Linux RCU effort.

Finally, I am indebted to Barry Leiner (in memoriam), Craig Partridge, Dale Goebel, Diane Lee, Lixia Zhang, and Phillip Kreuger for their help in getting me started down this road. After all, if you are not going to start a doctorate in the year 2000, when *are* you going to start on it?

Contents

Dedication . . . . . **iv**

Acknowledgements . . . . . **v**

Abstract . . . . . **xxi**

1 Introduction **1**

1.1 Where Has Moore's Law Gone Wrong? 2

1.2 SMMP Efficiency: The Effect on Real Code 3

1.3 What Can Be Done? 4

1.4 Returning to Basic Principles 6

1.5 Description of RCU 6

1.6 RCU Research Challenges 8

1.7 Contributions of this Dissertation 9

2 Related Work **11**

2.1 Synchronization in Uniprocessor Systems 11

2.2 Synchronization in SMMP Systems 16

2.2.1 Spinlocks 16

2.2.2 SMMP Semaphores 17

2.2.3 Performance of SMMP Locking 19

2.2.4 Hash-Table Mini-Benchmark 21

2.2.5 Code Locking 22

2.2.6 Reader-Writer Locking 23

2.2.7 Partitioning and Data Locking 25

2.2.8 Combining Data Locking and Reader-Writer Locking 27

2.2.9 Asymmetrical Reader-Writer Locking 28

2.2.10 Read-Side Performance Analysis 29

2.2.11 Write-Side Performance Experiment 33

2.2.12 Operating-System Partitioning 38

2.2.13 Simple Non-Blocking Synchronization 41

2.2.14 General Non-Blocking Synchronization 46
2.2.15 Transactional Hardware 55
2.2.16 Exploiting Semantics 56
2.2.17 Read-Mostly Linked-List Insertion 57
2.2.18 Read-Mostly Linked-List Removal 57
2.2.19 Read-Mostly Linked-List Removal and Reclaiming 59
2.2.20 Deferred Destruction 62
2.3 Discussion 64
2.3.1 Costs 65
2.3.2 Principles 65
2.3.3 Attributes 67

3 RCU Overview 71
3.1 Introduction to RCU 71
3.2 How RCU Solves Concurrency Problems 73
3.2.1 Readers and Writers 73
3.2.2 Multiple Versions 76
3.2.3 Writer-Writer Synchronization 78
3.2.4 Reader-Writer Synchronization 79
3.2.5 Weak Memory-Consistency Semantics 82
3.3 Conceptual Overview of RCU 83
3.3.1 RCU Glossary 83
3.3.2 RCU Design Tradeoffs 87
3.4 Examples 90
3.4.1 RCU Applied to Reader-Writer Locking 90
3.4.2 RCU Use of Deferred Destruction 93
3.5 RCU Analogies and Design Patterns 94
3.6 Discussion 95

4 Implementing RCU 99
4.1 RCU API 100
4.1.1 Read-Side RCU API 100
4.1.2 Write-Side RCU API 102
4.1.3 List-Manipulation RCU API 104
4.2 Implementing Grace-Period Detection 107
4.3 Inducing Quiescent States 108
4.3.1 Simple Induced Quiescent States 109
4.3.2 Induced Quiescent States With Batching (batch) 111

4.4 Observing Naturally Occurring Quiescent States . . . 113
4.4.1 Counters and Barrier (rcu-ltimer) . . . 113
4.4.2 Counter Ring (rcu-sched) . . . 123
4.4.3 Token Ring with Preemption (K42) . . . 130
4.5 Single-CPU RCU Implementation . . . 132
4.6 Discussion . . . 132
4.6.1 RCU API Discussion . . . 133
4.6.2 RCU Infrastructure Discussion . . . 134

5 Design Patterns for RCU . . . 137
5.1 Example Algorithm . . . 138
5.2 RCU Design Patterns . . . 138
5.2.1 Forces . . . 141
5.2.2 Non-RCU Locking Design Patterns . . . 144
5.2.3 Pure RCU . . . 147
5.2.4 RCU Existence Locks . . . 151
5.2.5 Reader-Writer-Lock/RCU Analogy . . . 153
5.2.6 RCU Readers With NBS Writers . . . 158
5.3 Patterns for Transforming Algorithms to RCU . . . 159
5.3.1 Forces . . . 160
5.3.2 Mark Obsolete Objects . . . 161
5.3.3 Substitute Copy For Original . . . 163
5.3.4 Impose Level Of Indirection . . . 168
5.3.5 Ordered Update With Ordered Read . . . 169
5.3.6 Global Version Number . . . 171
5.3.7 Stall Updates . . . 175
5.4 Discussion . . . 177
5.4.1 Index to Locking Design Patterns . . . 177
5.4.2 Index to Transformational Patterns . . . 178

6 Selected Applications of RCU Design Patterns . . . 179
6.1 System V IPC . . . 182
6.1.1 Semaphore Data Structures . . . 182
6.1.2 Semaphore Removal . . . 184
6.1.3 Semaphore Lock Acquisition . . . 185
6.1.4 Semaphore Deferred Deletion . . . 187
6.1.5 Semaphore Array Expansion . . . 188
6.1.6 Semaphore Operation . . . 191

6.1.7 Semaphore Discussion . . . 191
6.2 Linux Directory-Entry Cache . . . 195
6.2.1 Visual Overview of dcache . . . 195
6.2.2 Applying RCU to dcache . . . 203
6.2.3 dcache Discussion . . . 208
6.3 RCU Synchronizing With NMIs . . . 211
6.4 RCU and Module Race Reduction . . . 213
6.5 Incremental Use of RCU on tasklist Locking . . . 213
6.6 Scalable FD Management . . . 218
6.7 K42 Hash Tables . . . 221
6.8 Other RCU Usage . . . 227
6.8.1 VM/XA . . . 227
6.8.2 DYNIX/ptx . . . 227
6.8.3 RCU Use in K42 . . . 228
6.8.4 RCU Use in Linux . . . 229
6.9 Discussion . . . 234

7 Analytical Comparison of RCU and Locking . . . 235
7.1 Low-Contention Analytic Methodology . . . 235
7.1.1 Memory-Latency Model . . . 236
7.1.2 Conditions and Assumptions . . . 238
7.1.3 Procedural Details . . . 239
7.2 Low-Contention Derivations . . . 241
7.2.1 Derivation for Spinlock . . . 241
7.2.2 Derivation for Distributed Reader-Writer Spinlock . . . 243
7.2.3 Derivation for RCU . . . 250
7.2.4 Comparison . . . 254
7.3 Discussion . . . 265

8 Measured Comparison of RCU and Locking . . . 269
8.1 Comparison to Locking . . . 269
8.2 Comparison to Analytic Results . . . 271
8.3 Evaluation of Techniques for Identifying Grace Periods . . . 277
8.3.1 RCU Complexity . . . 277
8.3.2 RCU Overhead When Idle . . . 278
8.3.3 Grace-Period Latency . . . 279
8.3.4 RCU Overhead When In Use . . . 282
8.4 Discussion and Future Scenarios . . . 284

8.4.1 Uniprocessor Über Alles . . . 285
8.4.2 Multithreaded Mania . . . 286
8.4.3 More of the Same . . . 286
8.4.4 Crash Dummies Slamming into the Memory Wall . . . 287
8.4.5 Discussion of Future Scenarios . . . 287

9 Conclusions and Future Work . . . 289
9.1 Summary and Conclusions . . . 289
9.1.1 Challenges to Traditional Synchronization Mechanisms . . . 290
9.1.2 Definition of RCU . . . 291
9.1.3 Relation of RCU to Traditional Synchronization Mechanisms . . . 292
9.1.4 Analytic and Empirical Performance Evaluation . . . 292
9.1.5 Generality of RCU Via Design Patterns . . . 293
9.1.6 RCU Has Practical Value . . . 295
9.1.7 Summary . . . 295
9.2 Future Work . . . 296
9.2.1 RCU Infrastructure . . . 296
9.2.2 RCU Design Patterns . . . 300
9.2.3 RCU and Non-Blocking Synchronization . . . 300
9.2.4 RCU Uses in the Linux Kernel . . . 301
9.2.5 Suitability of RCU to Other Environments . . . 303
9.2.6 RCU Performance . . . 303
9.2.7 RCU Semantics . . . 306
9.3 Concluding Remarks . . . 306

Bibliography . . . 308

A Historical SMMP CPU Performance . . . 321

B Memory Ordering Issues . . . 322

C Additional RCU Implementations . . . 326
C.1 Induced Quiescent States With Batching (rcu-taskq) . . . 326
C.2 Further Leveraging of Quiescent States . . . 329
C.2.1 rcu-krcud . . . 330
C.2.2 X-rcu . . . 332
C.2.3 rcu-poll . . . 336
C.3 Preemption in Linux RCU (rcu-preempt) . . . 343

D RCU Performance on Large Hash Tables **346**

Biographical Note **350**

List of Tables

2.1 700 MHz P-III Operation Costs 32
2.2 Attributes of Synchronization Mechanism 69

3.1 Reader-Writer Locking and RCU 92
3.2 Attributes of Deferred Destruction Mechanisms 97

4.1 RCU Implementations Inducing Quiescent States 134
4.2 RCU Implementations Observing Quiescent States 135

5.1 Reader-Writer-Lock/RCU Substitutions 156
5.2 Locking Design Pattern Force Index 177
5.3 RCU Transformational Pattern Index 178

6.1 RCU Usage in Linux 2.6.0-test1 181
6.2 DBT1 Database Benchmark Results (TPS) 194
6.3 semopbench Microbenchmark Results (seconds) 194
6.4 Semaphore Change in Lines of Code 194
6.5 Applying RCU to get_pid_list() 216
6.6 Applying RCU to get_pid_list() Helper Macros 217

7.1 Nomenclature for Lock Cost Derivation 239
7.2 Simple Spinlock Access-Type Probabilities and Latencies 242
7.3 Distributed Reader-Writer Spinlock Memory Layout 243
7.4 Unnormalized Probability Matrix for Distributed Reader-Writer Spinlock . 244
7.5 Trace Labels 249

8.1 RCU Implementation Complexity 278
8.2 RCU Idle Overhead 279
8.3 dcachebench Comparison 284

A.1 Historical SMMP CPU Performance 321

List of Figures

1.1 SMMP Efficiency for Sequent Computers 2

2.1 Instructions per Local Memory Reference for Sequent Computers 20
2.2 Example Hash Table 21
2.3 Globally Locked Read-Only Hash Table Performance 23
2.4 Globally Reader-Writer Locked Read-Only Hash Table Performance 24
2.5 Comparing rwlock and spinlock 26
2.6 Bucket-Locked Read-Only Hash Table Performance 27
2.7 Bucket-Reader-Writer-Locked Read-Only Hash Table Performance 28
2.8 brlock Read-Only Hash Table Performance 29
2.9 Locked Increment 30
2.10 Out of Order Locked Increment 30
2.11 Locked Increment With Memory Barriers 31
2.12 Hash Table Performance for Mixed Workload on Four CPUs 34
2.13 Hash Table Performance for Co-Located Lock 37
2.14 Split Counter Initial Cache Configuration 43
2.15 Split Counter First Increment 43
2.16 Split Counter Subsequent Increments 44
2.17 Split Counter Readout 44
2.18 Split Counter Increment After Readout 45
2.19 NBS Update: Conceptual View 48
2.20 Non-Blocking Push 49
2.21 Non-Blocking Pop 50
2.22 Atomic Insertion Into a Linked List 58
2.23 Atomic Removal From a Linked List 59
2.24 Atomic Deletion From a Linked List 60

3.1 Causal Ordering for Exclusive Locking 75
3.2 Causal Ordering for Reader-Writer Locking 75
3.3 Causal Ordering for RCU 76
3.4 RCU Multiple Version Handling 77

3.5 RCU Multiple Version Timeline 78
3.6 RCU Concepts 87
3.7 Search-List Data Structures 91
3.8 Internal Search Algorithm 91
3.9 RCU Destruction Determination 94

4.1 RCU API 100
4.2 RCU List API 101
4.3 Example Read-Side RCU Critical Section 102
4.4 Example Blocking Write-Side RCU Critical Section 104
4.5 Example Non-Blocking Write-Side RCU Critical Section 105
4.6 Example Read-Side Use of RCU List-Manipulation Primitives 107
4.7 Example Write-Side Use of RCU List-Manipulation Primitives 108
4.8 RCU Grace Period 109
4.9 Simple Grace-Period Detection 110
4.10 Non-Preemptible Grace-Period Detection 110
4.11 Non-Blocking Grace-Period Detection 112
4.12 RCU Callback Tracking (rcu-ltimer) 115
4.13 *rcu-ltimer* Barrier-Computation Variables 116
4.14 *rcu-ltimer* call_rcu() Implementation 117
4.15 *rcu-ltimer* Scheduler Instrumentation 117
4.16 *rcu-ltimer* Per-CPU Timer Instrumentation 118
4.17 *rcu-ltimer* RCU-Callback-Pending Check 118
4.18 *rcu-ltimer* Tasklet Scheduling 119
4.19 *rcu-ltimer* Callback Advancement 120
4.20 *rcu-ltimer* Callback Invocation 120
4.21 *rcu-ltimer* Starting New Grace Period 121
4.22 *rcu-ltimer* Check for Quiescent State 122
4.23 Overlapping Grace Periods for Counter Ring 124
4.24 Counter Ring and CPU Hotplug 125
4.25 *rcu-sched* call_rcu() Implementation 127
4.26 *rcu-sched* Scheduler Instrumentation, Part 1 127
4.27 *rcu-sched* Scheduler Instrumentation, Part 2 128
4.28 *rcu-sched* Idle Loop Instrumentation 129
4.29 *rcu-sched* rcu_batch_done() 129

5.1 Lookup-Table Element 138
5.2 Lookup-Table Search 139

5.3 Relationship Among Patternssss 140
5.4 RCU Existence Locks 152
5.5 RCU Read-Side Code 155
5.6 RCU Write-Side Locking 155
5.7 Mark Obsolete Object Reader 162
5.8 Mark Obsolete Object Search 163
5.9 Mark Object Obsolete Upon Deletion 164
5.10 Substituting a Copy For Original 166
5.11 Substituting a Copy For Original Read 167
5.12 Ordered Update With Ordered Read 170
5.13 Non-Blocking Global Version Number Update 173
5.14 Stallable Update 176
5.15 Stalling Updates 176

6.1 Semaphore Structures with Global Locking 183
6.2 Semaphore Structures with RCU 184
6.3 Semaphore Deletion 185
6.4 Detecting Semaphore Deletion 186
6.5 Freeing a Semaphore 188
6.6 Expanding the Array of Pointers to Semaphores 189
6.7 Semaphore Initial State 191
6.8 Semaphore Structures After Array Replacement 192
6.9 Semaphore Structures After Grace Period 192
6.10 DBT1 Database Benchmark Raw Results 193
6.11 Example Filesystem Tree 196
6.12 dcache Representation of Example Filesystem Tree 197
6.13 dentry Hash Table 198
6.14 Hard-Link Alias Chains 199
6.15 dentry State Diagram 201
6.16 Traversing Mountpoints 202
6.17 VFS Mount Tree 202
6.18 Lock-Free Pathname Segment Lookup 205
6.19 Pathname Segment Lookup Rename Race Resolution 206
6.20 Deferred Free of dentry Structures 207
6.21 RCU Callback Function for dentries 207
6.22 Renaming dentries 209
6.23 Multiuser Benchmark Performance 210

6.24 NMI Timer Stop Function . . . 212
6.25 oprofile Shutdown Code . . . 212
6.26 Module Unloading . . . 214
6.27 Task-List RCU Patch . . . 215
6.28 SPEC SDET Performance of Task-List RCU . . . 218
6.29 Expanding FD Array . . . 219
6.30 RCU Expanding FD Array . . . 220
6.31 FD Management Performance . . . 221
6.32 Lock-Free Hash Remove Operation . . . 224
6.33 Lock-Free Hash Remove Operation with Race . . . 225
6.34 PostMark Performance of Hash Table . . . 226

7.1 NUMA Memory Latency . . . 237
7.2 Costs of Simple Spinlock and Distributed Reader-Writer Spinlock . . . 249
7.3 Breakevens vs. Number of CPUs, λ Small . . . 255
7.4 Breakevens vs. Number of CPUs, λ=0.1 . . . 256
7.5 Breakevens vs. Number of CPUs, λ=1.0 . . . 256
7.6 Breakevens vs. Number of CPUs, λ=10.0 . . . 257
7.7 Breakevens vs. Number of CPUs, λ Large . . . 257
7.8 Breakevens vs. λ, Two CPUs . . . 258
7.9 Breakevens vs. λ, Four CPUs . . . 258
7.10 Breakevens vs. λ, Eight CPUs . . . 259
7.11 Breakevens vs. r, λ Small, Two CPUs . . . 260
7.12 Breakevens vs. r, λ=0.1, Two CPUs . . . 260
7.13 Breakevens vs. r, λ=1.0, Two CPUs . . . 261
7.14 Breakevens vs. r, λ=10.0, Two CPUs . . . 261
7.15 Breakevens vs. r, λ Large, Two CPUs . . . 262
7.16 Breakevens vs. r, λ Small, Four CPUs . . . 262
7.17 Breakevens vs. r, λ=0.1, Four CPUs . . . 263
7.18 Breakevens vs. r, λ=1.0, Four CPUs . . . 263
7.19 Breakevens vs. r, λ=10.0, Four CPUs . . . 264
7.20 Breakevens vs. r, λ Large, Four CPUs . . . 264
7.21 Breakevens vs. r, λ Small, Eight CPUs . . . 265
7.22 Breakevens vs. r, λ=0.1, Eight CPUs . . . 266
7.23 Breakevens vs. r, λ=1.0, Eight CPUs . . . 266
7.24 Breakevens vs. r, λ=10.0, Eight CPUs . . . 267
7.25 Breakevens vs. r, λ Large, Eight CPUs . . . 267

8.1 Two-CPU Hash Table Performance for Short-Grace-Period Mixed Workload 272
8.2 Four-CPU Hash Table Performance for Short-Grace-Period Mixed Workload 273
8.3 Two-CPU Hash Table Performance for Long-Grace-Period Mixed Workload 274
8.4 Four-CPU Hash Table Performance for Long-Grace-Period Mixed Workload 275
8.5 RCU Hash Table Breakevens on Two CPUs 276
8.6 RCU Hash Table Breakevens on Four CPUs 277
8.7 call_rcu() Latency Under dbench Load 280
8.8 call_rcu() Latency Under dbench Load (logscale) 281
8.9 RCU Performance on Chat Benchmark 283

B.1 Insert and Lock-Free Search 323
B.2 Why smp_read_barrier_depends() is Required 324
B.3 Safe Insert and Lock-Free Search 325

C.1 *rcu-taskq* call_rcu() Implementation 327
C.2 *rcu-taskq* process_pending_rcus() Implementation 328
C.3 *rcu-taskq* wait_for_rcu() Implementation 328
C.4 *rcu-taskq* krcud() Implementation 329
C.5 *rcu-krcud* call_rcu() Implementation 331
C.6 *rcu-krcud* Scheduler Instrumentation 331
C.7 *rcu-krcud* Timer Processing 331
C.8 *X-rcu* call_rcu() Implementation 332
C.9 *X-rcu* Scheduler Instrumentation 333
C.10 *X-rcu* Timer Processing 333
C.11 RCU Callback Flow 334
C.12 *X-rcu* rcu_process_callbacks() 334
C.13 *X-rcu* rcu_move_next_batch() 335
C.14 *X-rcu* rcu_check_quiescent_state() 337
C.15 *X-rcu* rcu_invoke_callbacks() 338
C.16 *X-rcu* rcu_reg_batch() 338
C.17 *rcu-poll* call_rcu() Implementation 339
C.18 *rcu-poll* Scheduler Instrumentation 339
C.19 *rcu-poll* Tasklet Body 340
C.20 *rcu-poll* rcu_prepare_polling() 341
C.21 *rcu-poll* rcu_polling() 341
C.22 *rcu-poll* rcu_completion() 342
C.23 *rcu-poll* rcu_invoke_callbacks() 342
C.24 rcu_preempt Per-CPU Counters 345

D.1 Large Hash Table Performance for Read-Only Workload 347
D.2 Large Hash Table Performance for Mixed Workload and Short Grace Period 348
D.3 Large Hash Table Performance for Mixed Workload and Long Grace Period 349

Abstract

Exploiting Deferred Destruction:
An Analysis of Read-Copy-Update Techniques
in Operating System Kernels

Paul E. McKenney

Supervising Professor: Jonathan Walpole

The Moore's-Law-driven performance of simple instructions has improved by orders of magnitude over the past two decades, but shared-memory multiprocessor (SMMP) synchronization operations have not kept pace. SMMP software uses synchronization operations heavily, thus suffering degraded performance and scalability. As a result, many traditional SMMP algorithms are now obsolete.

This dissertation presents read-copy update (RCU), a reader-writer synchronization mechanism in which read-side critical sections incur virtually zero synchronization overhead, thereby achieving near-ideal performance for read-mostly workloads. Write-side critical sections incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the synchronization-free read-side critical sections. In addition, writers use some mechanism, such as locking, to ensure orderly updates.

Readers provide a signal enabling writers to determine when it is safe to complete destructive operations, but this signal may be deferred, permitting a single signal operation to serve multiple read-side RCU critical sections.

These read-side signals are observed by a specialized garbage collector, which carries out destructive operations once it is safe to do so. Garbage collectors are typically implemented in a manner similar to a barrier computation. Production-quality garbage collectors batch destructive operations, amortizing signal-observation overhead over many updates.

Although RCU is not itself new, its use has been quite specialized. This dissertation rectifies this situation by showing how RCU can be implemented efficiently in operating system kernels, by demonstrating its system-level performance and complexity benefits, and by providing a set of design patterns that make RCU more generally applicable.

This dissertation compares RCU to traditional synchronization mechanisms, including locking and non-blocking synchronization, using both analytic and empirical methods. The empirical methods include both informal micro-benchmarks and formal system-level benchmarks. These benchmarks show performance benefits ranging from tens of percent to an order of magnitude and little or no increase in code complexity.

Finally, this dissertation demonstrates that RCU has practical value by (1) outlining its use in several production systems, two of which have seen extensive datacenter use, one of which this author designed and implemented, and (2) documenting its widespread use in the Linux 2.6 kernel.

Chapter 1

Introduction

Although computer system performance has increased by several orders of magnitude over the past few decades, this Moore's-Law performance increase has not been so beneficial to shared-memory multiprocessor (SMMP) software, as shown by Figure 1.1. Each trace plots the ratio of the cost of the specified number of instructions to the cost of a spinlock-guarded critical section containing that same number of instructions, where contention is negligible and where both the lock and the data that it guards are contained in a single cache line. The instruction cost is estimated based on the published MIPS ratings for the older CPUs, and based on the clock frequency for newer CPUs. The cost of lock acquisition and release is estimated based on the cost of moving a line from one CPU's cache to another's. The raw data is displayed in Appendix A.1 on page 321.

This ratio is a measure of the efficiency of SMMP software for the specified critical-section size, measured in number of instructions. As can be seen in the figure, this efficiency has decreased precipitously since the early 1980s, so that SMMP algorithms that both performed and scaled extremely well two decades ago perform abysmally today. The Moore's-Law-induced changes in computer architecture have rendered such algorithms obsolete.

Since Moore's Law has brought huge benefits to computing and to society in general, it is only natural to ask why it has served SMMP software so poorly.

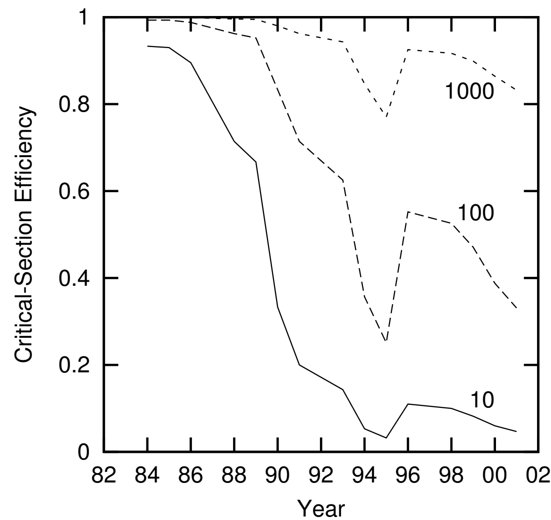


Figure 1.1: SMMP Efficiency for Sequent Computers

1.1 Where Has Moore's Law Gone Wrong?

The trend displayed in Figure 1.1 is nothing new, nor has it gone unnoticed [6, 17, 32, 38, 68, 73, 88, 117]. As discussed in these references, this trend has a number of underlying causes:

1. Electrical and speed-of-light effects have confined the sharpest rise in clock frequency to on-chip components. Connections between chips run at much lower clock frequencies, limiting bandwidths and degrading latencies for the chip-to-chip data paths connecting CPUs to each other and to memory.

2. Increasing memory size increases the depth of the address-decoding logic, thus increasing memory latency.

3. Increased memory latency is addressed by use of larger caches and larger numbers of levels of cache. Although these caches greatly improve average memory latency, they actually *degrade* worst-case latency, because each level in the hierarchy imposes a latency penalty, regardless of whether the access is satisfied from DRAM or from

another CPU's cache.

4. Some of the CPU core-clock frequency increase has been enabled by increasing the number of stages in CPU pipelines and by heuristic measures such as branch prediction. Although these longer pipelines increase best-case instruction rate, they also increase the penalty for branch mispredictions and other events that stall the pipeline. The increased penalty is due to the larger amount of partially executed instructions delayed or discarded when the longer pipeline is stalled.

5. Both increased memory latency and the overhead of pipeline stalls is partially addressed via superscalar CPUs, which do out-of-order and speculative execution. However, many SMMP algorithms do not tolerate such reordering, and they must therefore use special "memory barrier" instructions to enforce the needed ordering. Unfortunately, these memory-barrier instructions are quite expensive.

The result of these effects has been that although the single-threaded instruction-execution performance of CPUs has increased dramatically, the performance of CPU-to-CPU, CPU-to-memory, and pipeline-stall operations has not kept pace. However, these are extremely low-level micro-architectural issues. Have they really had significant impact on the performance of real-world software running on real-world computer systems?

1.2 SMMP Efficiency: The Effect on Real Code

The decrease in SMMP efficiency has been dramatic, as can be seen from Figure 1.1. In fact, Chapters 2, 7, and 8 will show that the performance of the synchronization operations traditionally used by SMMP software critically depends on the performance of CPU-to-CPU, CPU-to-memory and pipeline-stall operations that have been left in Moore's Law's wake.

In particular, Section 2.2 describes how traditional SMMP synchronization primitives rely on atomic instructions to update shared data. Because the data is shared, each such update will likely need to communicate with the CPU that performed the previous update, unless great care is invested into increasing temporal locality. The performance of such

CPU-to-CPU communication is quite slow, and is actually degraded by the same caches that are used to improve average memory-access latency. In addition, as noted earlier, most SMMP algorithms require that operations be ordered,[1] and the memory-barrier instructions used to accomplish this are quite expensive.

Recent hardware research has uncovered methods of optimizing the execution of critical sections, as is discussed in Section 2.2.15 on Page 55. However, all such hardware improvements are currently research prototypes, and will require some time to appear in commodity products, if in fact they ever do make their way into commodity products. Therefore, it makes sense to look for new ways of writing SMMP software so as to avoid these expensive operations.

1.3 What Can Be Done?

Since the trends outlined in the previous sections are not new, one would expect that much work has already been done to address them. For example, one approach suggested directly by Figure 1.1 is simply to increase the size of critical sections, perhaps by merging adjacent sections. This approach has the benefit of decreasing complexity, but unfortunately typically also decreases scalability [66].

The opposite approach of partitioning critical sections across different data elements, or "data locking", has met with substantial success, as described in Section 2.2.7 on Page 25. However, even though data locking can greatly reduce lock contention, it does nothing to reduce the use of expensive atomic instructions and memory barriers.

In principle, read-mostly data structures can benefit from the greatly decreased levels contention provided by reader-writer locks, but, in practice, as will be shown in Section 2.2.6 on Page 23, memory latency often overwhelms the read-side parallelism benefits of reader-writer locking. Specialized asymmetric reader-writer locks can be used in some cases, but Section 2.2.9 on Page 28 will show how far short of ideal performance they fall, and Chapters 7 and 8 will demonstrate their shortcomings in update-intensive situations.

[1]For example, it is not permissible to allow the operations following a lock acquisition to be executed before the lock acquisition completes. After all, the whole purpose of acquiring the lock is to protect those operations, and it cannot very well do so if they have been reordered to precede the lock acquisition.

Non-blocking synchronization (NBS) has seen much research activity over the past decade, and in fact Section 2.2.13 on Page 41 will show that simple forms of NBS have seen widespread use, though only in very specialized situations. Section 2.2.14 on Page 46 describes the difficulties that NBS has faced in more general situations, namely, that although it does eliminate lock contention, it introduces a similar form of contention-induced performance degradation due to repeated retries. In addition, NBS does nothing to reduce performance degradation due to the pipeline-stall and memory-latency overhead incurred by memory barriers and by atomic updates of shared memory. Nonetheless, many of the ideas put forth in NBS research are valuable in their own right, as will be shown in Chapter 5.

Section 2.2.16 on Page 56 will describe how the semantics of underlying data and operations may be exploited to increase performance and parallelism. In fact, the NBS split-counter example to be described in Section 2.2.13 on Page 2.2.13 exploits the commutative law of addition for exactly this purpose. However, these methods are specialized to specific situations, and thus are not generally applicable.

A number of researchers have noted that read-only accesses to read-mostly data structures may entirely dispense with synchronization operations if writers defer destruction [1, 30, 39, 53, 54, 56, 61, 62, 81, 86, 87, 91, 93]. Dispensing with synchronization operations in the read-only case is important: given that the concurrent readers need not communicate amongst themselves, the expensive synchronization operations are unnecessary, at least in principle. However, as will be discussed in Section 2.2.20 on Page 62, most of these researchers failed to present an efficient and robust mechanism for determining how long to defer destruction in SMMP environments lacking a garbage collector, but with long-running processes. Even those researchers who provided such a mechanism either:

1. imposed memory-barriers on readers, with attendant pipeline-stall overhead [86, 87];
2. implemented a solution that ran only on a single CPU architecture, and that was exploited only in an ad-hoc fashion [81, 108]; or
3. exploited their solution for only one of the many purposes to which it could profitably be put [30].

The fact that so much work has provided so little benefit for read-mostly SMMP situations raises the question of whether we need to return to basic principles in order to better understand exactly what benefit can be provided.

1.4 Returning to Basic Principles

Section 2.3.2 on Page 65 will abstract three basic principles from the related work surveyed in Chapter 2:

1. Avoid expensive operations, namely those incurring pipeline-stall and memory-latency overheads.
2. Architect, design, and implement algorithms that permit fully parallel execution, thereby avoiding contention.
3. Architect, design, and implement algorithms that meet software-engineering needs, including simplicity, resilience against denial-of-service attacks, and tolerance for incremental adoption.

This dissertation will apply these principles to read-mostly data structures, such as those representing hardware and software configuration in operating-system kernels. Doing so results in significant performance improvements ranging from tens of percent to an order of magnitude with little or no increase in complexity. These benefits are obtained by refining the definition and use of a deferred-destruction technique, namely read-copy update,[2] which will henceforth be abbreviated as RCU.

1.5 Description of RCU

RCU is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur zero

[2]The name “read-copy update” originated from its ability to permit readers to run concurrently with updates, as long as the updates use copy operations so as to appear atomic to the readers. This idiom (or “design pattern”) is described in more detail in Section 5.3.3 on Page 163.

synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance for read-only workloads on most architectures, and can greatly simplify the structure of some algorithms [30]. Write-side critical sections must therefore incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to handle concurrent updates.

Readers must provide a signal enabling writers to determine when it is safe to complete destructive operations, but this signal may be deferred, permitting a single signal operation to serve multiple read-side RCU critical sections. RCU typically signals writers by non-atomically incrementing a local counter, which is an extremely inexpensive operation.

These read-side signals are observed by a specialized distributed garbage collector, which uses a lazy barrier or a combining tree to sense the reader signals, and carries out destructive operations once all readers have signalled that it is safe to do so. Garbage collectors are typically implemented in a manner similar to a barrier computation, or, on NUMA systems, a combining tree. Production-quality garbage collectors batch destructive operations, so as to amortize their overhead over many write-side update operations.

RCU provides concurrent reads: because readers do not use any synchronization mechanism, there is no way for readers to avoid concurrency. For the same reason, RCU provides concurrent reads and writes. RCU does not specify whether writers may run concurrently with each other; write-side concurrency depends instead on the chosen write-side synchronization mechanism. The fact that RCU read-side critical sections use no synchronization mechanisms means that there is no overhead due to pipeline stalls, memory latency, contention, or locking for readers. Write-side overhead depends on the chosen write-side synchronization mechanism, but contention is reduced due to the fact that readers do not use any synchronization mechanisms.

In practice, it is also important that the read-side signalling occur reasonably frequently, since during a time interval in which a given CPU fails to provide such a signal, pending destructive operations cannot be carried out. If this condition persists, all available memory could be consumed tracking these pending destructive operations.

RCU is not new. The first mention of a similar mechanism occurred in 1980 [56], an early implementation was used in production in the late 1980s [39], and more efficient implementations followed in the 1990s [108, 81, 30]. Given RCU's long history, what research challenges could possibly remain?

1.6 RCU Research Challenges

The first challenge is to identify the deficiencies of traditional synchronization mechanisms, for if there are no compelling deficiencies, then there is no reason to adopt a new synchronization mechanism. Chapter 2 takes up this task.

The second challenge is to articulate clearly the conceptual basis for RCU, showing how it operates and what form of synchronization it provides. This challenge is met by Chapter 3.

The third challenge is to provide an efficient and robust implementation of RCU that meets the simplicity and portability needs of general-purpose operating systems. The efficiency of the RCU implementation is critically important, because the write-side overhead of a wasteful implementation could overwhelm the read-side savings. A description of six candidate implementations for the Linux kernel, written by various Linux-community members, may be found in Chapter 4. One of these candidates was accepted into the Linux 2.6 kernel.

In its raw form, RCU is quite difficult to use because readers can see stale data as well as inconsistent views of non-atomically updated data. The use of RCU in its raw form is restricted to very specialized situations. The fourth challenge is therefore to make RCU easier to use and more generally applicable, which is a software engineering challenge. Chapter 5 attacks this problem by presenting design patterns showing how RCU may be used, along with a transformational design pattern language that transforms algorithms that are unable to tolerate RCU's staleness and inconsistency properties into forms that are able to tolerate these properties. Chapter 6 presents uses of these patterns taken from VM/XA, DYNIX/ptx, Tornado, K42, and the Linux 2.6 kernel.

The final challenge is evaluating the performance and complexity of RCU. This challenge is taken up using analytic techniques in Chapter 7 and using empirical techniques with mini-benchmarks in Chapter 8. Chapter 6, along with its presentation of design-pattern usage, includes empirical measurements of RCU performance using formal benchmarks and simple empirical measurements of RCU code complexity.

These five challenges by no means exhaust RCU's research possibilities; areas of future work are presented in Section 9.2 on Page 9.2.

1.7 Contributions of this Dissertation

The contributions of this dissertation include:

1. A survey of performance-related changes in computer-system architecture and of the consequent challenges to the synchronization mechanisms traditionally used by software running on SMMP systems.

2. The definition of a solution, namely RCU, to a set of concurrency problems stemming from these changes and challenges.

3. A presentation of the relationship between RCU and traditional synchronization mechanisms.

4. A demonstration of the performance benefits, ranging from tens of percent to an order of magnitude, that RCU can provide with little or no increase in complexity. This demonstration uses analytic methods, informal micro-benchmarks, and formal system-level benchmarks.

5. A set of design patterns that permit RCU to be profitably applied to a wide range of synchronization problems, along with examples where these patterns have been used, both by myself and by others.

6. A demonstration of the practical value of RCU, as evidenced by its use in several production systems, one of whose RCU infrastructure this author designed and

implemented. This practical value is further evidenced by RCU's acceptance and widespread use within the Linux 2.6 kernel.

Chapter 2 discusses related work, evaluating this work on the basis of an in-depth treatment of synchronization's performance and complexity problems. Chapter 3 presents an overview of RCU, including a short glossary and description of concepts. Chapter 4 describes a number of algorithms that have been used to implement deferred destruction, one of which has been accepted into the Linux 2.6 kernel. Chapter 5 extends a locking-design pattern language [67] to incorporate RCU, and also presents an additional pattern language which transforms existing algorithms into a form more compatible with RCU. Chapter 6 describes how RCU has been used in various operating-system kernels, evaluates its performance, and shows what patterns were used in each case. Chapter 7 analytically compares the performance of RCU to that of various locking techniques and Chapter 8 uses empirical measurements to compare alternative implementations of RCU infrastructure as well as to compare the performance of RCU, both to that of various locking techniques and to that predicted by the analysis. Finally, Chapter 9 presents conclusions and areas for further study.

Chapter 2

Related Work

A thread may be thought of as part of a running program, but it is useful to define this term in a broad sense, so that it includes any executing entity such as a process, task, coroutine, interrupt handler, or signal handler. In a group of cooperating threads, the results of each individual thread's computation might either affect or be affected by those of the other threads.

One problem arising in concurrent systems of cooperating threads is that of providing orderly access and updates to data. This problem, called synchronization, has received a great deal of attention over the past several decades. CPU designers provided synchronization operations at least four decades ago, and researchers have published many shelves full of textbooks, journal articles, and conference papers on hardware and software synchronization mechanisms.

This chapter reviews the major areas of synchronization research and identifies some key gaps, which this dissertation addresses. This chapter first focuses on synchronization in uniprocessor systems in Section 2.1, and then on synchronization in multiprocessor systems in Section 2.2. Section 2.3 provides summary and discussion.

2.1 Synchronization in Uniprocessor Systems

Interrupt handlers and preemptive scheduling introduced the need for synchronization, even on uniprocessors, many decades ago.

Consider, for example, a system running a single user thread that repeatedly computes some values, then writes these values to disk. The simplest implementation would

compute a given set of values, initiate the I/O that writes these values to disk, wait until this I/O completes, then start computing the next set of values. The problem with this implementation is that the disk is idle while the CPU computes a given set of values, and the CPU is idle while that set of values is written to disk. Because disks are slow devices, with response times of milliseconds, literally millions of instructions are wasted by the CPU while waiting for disk I/O to complete.

The standard solution to this problem is overlapped I/O, where the CPU computes one set of values while the previous set of values is being written to disk. However, this solution requires that there be a way of determining when the previous I/O completes, so that the corresponding memory may be reused. This determination is typically made via an *interrupt*, which causes a predetermined *interrupt handler* to be invoked. This interrupt-handler invocation may be thought of as an unsolicited function call that preempts the currently running thread, resuming it upon return. Such an interrupt handler could free the memory upon I/O completion, thus permitting its reuse.

However, this solution immediately presents another problem. What if the running thread was itself allocating or freeing memory at the time that the interrupt occurred? If the memory allocator's data structures are in an inconsistent state at the time of the interrupt, the interrupt handler's attempt to free memory could result in a system crash, or worse.[1]

One solution is for the thread to disable interrupts while allocating or freeing memory. The interrupt handler will therefore never be invoked while the memory allocator's data structures are in an inconsistent state, thereby ensuring the memory allocator's integrity. Disabling interrupts has been a standard technique for many decades; The 1965 CDC 3300 [20, 89] was capable of disabling interrupts, and it was not the first computer to do so. However, disabling interrupts introduces yet another problem, namely that it can delay invocation of the interrupt handler or result in an interrupt being "lost", so that a pair of consecutive interrupts results in the handler being invoked only once.

One the one hand, if delay is acceptable, then the standard technique is for the interrupt

[1] What could possibly be worse than a system crash? Intermittent silent data corruption! Everything looks fine, but the answer is wrong. Sometimes. For no apparent reason.

handler to check the device state before returning, thus handling any number of interrupts that might have been lost. On the other hand, if delay is unacceptable, another well-known solution is for the thread and the interrupt handler to use atomic instructions to manipulate shared data structures. This use of atomic instructions, such as the CDC 3300's "Replacement Add" or the x86 locked exchange instructions [51], result in all manipulations of the memory allocator's data structure being atomic, so that the interrupt handler always sees this data structure in a consistent state. However, this approach solves the problem efficiently only for data structures amenable to atomic manipulation.

Operating-system and device-driver writers have used interrupt disabling as a synchronization mechanism for many years. However, this mechanism must be used carefully to avoid interfering with critical system services, such as thread scheduling and network protocol handling, that depend on timer interrupts. For this reason, untrusted user threads cannot be allowed to disable interrupts.

Suppressing Preemption

An alternative approach is to provide synchronization primitives to user threads. One way to accomplish this is to provide a system call that sets a bit in the kernel that prevents the scheduler from preempting the currently running thread. All I/O and timer interrupt handlers would then run normally, permitting the kernel to properly interact with I/O hardware and networking protocols, while still preventing any other user threads from accessing data that the currently running thread has left in an inconsistent state. This approach still permits a malicious or buggy user thread to disable preemption for extended time periods. However, the operating system can periodically regain control and kill an offending thread.

Given a preemption-suppressing system call, a typical method of safely incrementing a variable is as follows:

1. Suppress preemption.

2. Load the variable into a register.

3. Increment the register.

4. Store the register into the variable.

5. Enable preemption.

Once a given thread executes step 1 of this procedure, no other threads may run. The three-step increment of the variable thus appears atomic to these other threads, as required. Step 5 then re-enables preemption, permitting other threads to run. If all threads use the above procedure, the variable will be atomically incremented, so that no increments are lost.

Although unconditional suppression of preemption is very simple, easy to use, and efficient, it can also be used in a denial-of-service attack, where a malicious thread disables preemption and then enters an infinite loop. As noted earlier, such attacks could be addressed by terminating threads that suppress preemption for too long, but repeated attacks could still consume much of the system's resources.

Uniprocessor Semaphores

One way to prevent such abuse is for the operating system to instead provide locking primitives to user threads. The prototypical example of such locking primitives are semaphores [25], which are implemented, for example, by the Unix System V semaphore primitive [139].

Although System V semaphores are quite flexible and feature-rich, they are frequently used as simple locks, for example to guard a Hoare monitor [47]. For sake of simplicity, this chapter will consider implementations of this subset of semaphore functionality.

To use a semaphore, one of the user threads would execute a `semget()` system call to allocate a semaphore. This semaphore is owned by the user running the set of cooperating threads, who can set permissions to prevent other users from interfering with the semaphore, thereby preventing denial-of-semaphore attacks.

Once a semaphore is created, threads with sufficient permissions may use the `semop()` system call to acquire and release the semaphore. If one thread holds the semaphore, other threads attempting to acquire it will block within the kernel until it has been released.

The kernel represents the semaphore with a data structure that includes: (1) an indication of whether the semaphore is available, and (2) a queue of threads waiting for the semaphore to become available. The operating-system kernel disables preemption if needed (for example, by disabling interrupts) while manipulating this data structure in order to maintain its integrity. This disabling of preemption is unconditional, which is acceptable because the kernel code base is trusted to restore preemption in a timely manner. Furthermore, in-kernel semaphore implementations need to disable preemption only while acquiring and releasing the semaphore, rather than for the full duration of the critical section.

The simplified version of the semaphore acquisition code might be implemented as follows in the kernel:

1. Disable interrupts (which has the side effect of disabling preemption).
2. While the semaphore is held by some other thread, repeat the following steps:
   (a) Add this thread to the tail of the queue of threads waiting for this semaphore.
   (b) Reenable interrupts.
   (c) Switch context to some other thread.
   (d) (Get here when the thread is awakened. It is not guaranteed that this thread will successfully acquire the semaphore, since some other thread may have acquired it between the time that the semaphore was released and the time that this thread started running.)
   (e) Disable interrupts.
3. Mark the semaphore as held by this thread.
4. Reenable interrupts.

Note that threads attempting to acquire the semaphore when it is already held will block, thereby returning control to the scheduler, which has the advantage of allowing the thread holding the semaphore to run, which hopefully allows that thread to release it in a timely fashion.

Releasing the semaphore would be implemented as follows:

1. Disable interrupts.
2. Mark the semaphore as released.
3. If the queue of threads waiting for this semaphore is nonempty:
    (a) Remove the first thread from the queue of threads waiting for this semaphore.
    (b) Awaken the newly removed thread.
4. Reenable interrupts.

Thus, the high-level semaphore synchronization mechanism is implemented in a layered fashion, in terms of a lower-level preemption-disabling mechanism. This means that semaphores may be implemented simply, efficiently, and safely on hardware with minimal synchronization features. All that is required is the ability to disable interrupts, which is available on all modern microprocessors.

2.2 Synchronization in SMMP Systems

The interrupt-disabling technique described in the previous section will fail on SMMP systems, since a pair of threads running on different CPUs could simultaneously acquire the semaphore. Both threads would believe that they held the semaphore, which could result in the application crashing, or, worse yet, silently corrupting the shared data. The approach of simply refusing to support SMMP systems is becoming less and less viable, given the advent of hardware multithreading in commodity microprocessors.

SMMP operating-system kernels clearly cannot rely solely on interrupt or preemption disabling; implementing the semaphore primitives on SMMP systems also requires some way of excluding other CPUs from a given critical section.

2.2.1 Spinlocks

One approach is to use spinlock primitives to protect the semaphore. The difference between a spinlock and a semaphore is that a thread waiting to acquire a spinlock will

busy-wait instead of blocking. Because spinlocks do not block, they do not need to interact with the scheduler, and can thus be implemented in terms of atomic instructions.

For example, the Linux kernel bases its spinlocks on the atomic decrement primitive. The spinlock is initialized to 1, and spinlock acquisition proceeds roughly as follows (the actual implementation is more complex for performance reasons):

1. Atomically decrement the spinlock value.
2. If the value was negative, someone else holds the spinlock, so busy-wait (or "spin") as follows:
   (a) If the spinlock value is now positive, restart the acquisition procedure from step 1.
   (b) Otherwise, restart the spin from step 2a.

If multiple CPUs simultaneously attempt to acquire the spinlock, the one that decrements the spinlock to zero will hold the spinlock, while the rest will spin in step 2 until the spinlock is released. The spinlock is released by setting its value back to 1.

Spinlocks may also be based on a number of other types of special instructions [26, 49], or even on normal loads and stores [24, 57], though this latter approach is quite complex and slow.

If a given CPU holds the spinlock for an extended period of time, the rest of the CPUs will waste a large amount of time in step 2 spinning for the spinlock. In such cases, it would be better to use a semaphore so that waiting threads could block, permitting other work to get done. In contrast, spinlocks are best for protecting short critical sections, where the duration of the critical section is shorter than twice the time required for the scheduler to switch from one thread to another.

2.2.2 SMMP Semaphores

SMMP semaphores may be constructed using spinlocks. In this implementation, the semaphore data structure therefore contains a spinlock in addition to the semaphore-held

indicator and the thread-waiting queue. SMMP semaphore acquisition is quite similar to that of the uniprocessor case, with the interrupt disabling replaced by spinlock acquisition:

1. Acquire the semaphore's spinlock.
2. While the semaphore is already held, repeat the following steps:
   (a) Add this thread to the tail of the queue of threads waiting for this semaphore.
   (b) Release the semaphore's spinlock.
   (c) Switch context to some other thread.
   (d) (Get here when the thread is awakened. It is not guaranteed that this thread will successfully acquire the semaphore, since some other thread may have acquired it between the time that the semaphore was released and the time that this thread started running.)
   (e) Acquire the semaphore's spinlock.
3. Mark the semaphore as held.
4. Release the semaphore's spinlock.

Processes waiting on the semaphore block in step 2c, allowing the CPUs to run other threads in the meantime.

Similarly, releasing a semaphore closely resembles the uniprocessor case:

1. Acquire the semaphore's spinlock.
2. Mark the semaphore as released.
3. If the queue of threads waiting for this semaphore is nonempty:
   (a) Remove the first thread from the queue of threads waiting for this semaphore.
   (b) Awaken the newly removed thread.
4. Release the semaphore's spinlock.

The spinlock guarantees that only one CPU at a time may manipulate the semaphore data structure. This is another example of a layered implementation of a synchronization primitive: the semaphore implementation is layered on top of the simpler spinlock implementation.

Both the spinlock and the semaphore provide the same function: mutual exclusion. However, there are performance differences. As noted earlier, spinlocks that are held for extended periods can result in the squandering of much CPU time spinning on the lock. On the other hand, semaphores inflict their own performance penalties in other situations. Because the semaphore acquires and releases a spinlock at least once during both semaphore acquisition and release, semaphores will be at least twice as expensive as spinlocks when there is no contention. Furthermore, the context-switch operation is itself relatively expensive. If only mild contention is expected, and if the critical sections take less time to execute than is required to do a context switch, it will be more efficient to spin than it will to context switch.

Because switching context is relatively expensive and critical sections are frequently quite short, a large fraction of production-OS critical sections are protected by spinlocks rather than by semaphores. The remainder of this chapter will therefore focus on spinlocks.

2.2.3 Performance of SMMP Locking

Moore's Law has held for more than 40 years, during which time integrated-circuit transistor counts have doubled roughly every 12-18 months [88]. CPU clock rates have also increased exponentially over the past few decades. Given this wildly increasing computing capability, one might expect that synchronization primitives would be extremely cheap, so that any performance-related discussion of them would be a waste of time.

However, this is not the case. These primitives are subject to the following four types of overhead, in increasing order of severity on Pentium-III™ CPUs:

1. Instruction-execution overhead.

2. Pipeline-stall overhead, resulting from things like branch mispredictions and memory barriers.

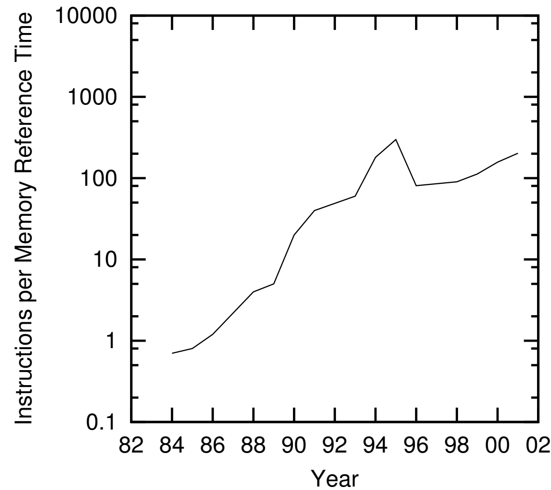


Figure 2.1: Instructions per Local Memory Reference for Sequent Computers

3. Memory latency, incurred when data moves among memory and the CPUs' caches. Note that from a CPU-architecture viewpoint, the caches are part of memory, being simply an optimization for spatial and temporal memory-reference locality. The CPU used in this dissertation has about the same DRAM and CPU-to-CPU latencies, however, future CPUs may require that these two types of memory latency be treated separately.

4. Contention effects, such as spinning and context switching.

Figure 2.1 plots the ratio of memory latency to instruction-execution overhead for the past two decades, showing that improvements in memory latency have not kept up with Moore's-law improvements in instruction-execution rate. The downward blip at 1996 was due to the introduction of so-called "glueless MP", which greatly decreased the electrical distance between CPUs. The trend since 1996 has continued inexorably upwards.

What are the implications of Figure 2.1 on real program performance? The following sections provide application-level data, namely the overhead of searching a hash table in parallel on a 4-CPU 700MHz Pentium-III computer system.

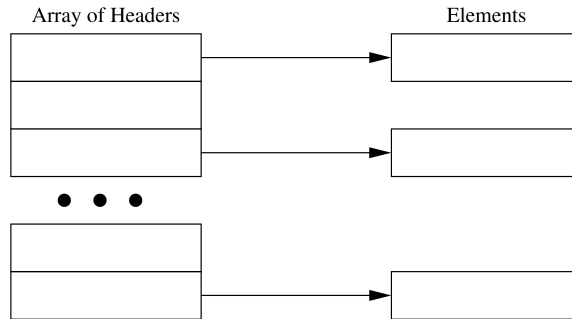


Figure 2.2: Example Hash Table

2.2.4 Hash-Table Mini-Benchmark

The preceding section showed that memory-latency overhead has increased dramatically over the past two decades. This increase in overhead decreases performance of SMMP synchronization primitives, which might well cause SMMP systems to be slower than uniprocessor systems. Because the whole point of using multiple CPUs is to increase performance, it is necessary to compare synchronization mechanisms to see which are best able to overcome SMMP overheads. Because operating-system kernels make extensive use of hash tables, a simple chained hash-table mini-benchmark is an appropriate vehicle for making these comparisons.

This hash table consists of a dense array of pointers that reference the hash-chain list for the corresponding bucket, as shown in Figure 2.2. Various synchronization and mutual-exclusion mechanisms will be compared using mixtures of read-only searches, insertions, and deletions. In order to simplify analysis, deletions and insertions will always be performed in pairs, so that the number of elements in the hash table remains constant. However, each such deletion and insertion is performed as a separate operation in order to faithfully mimic real applications, which would not normally remove an element and then immediately re-insert it as a single combined operation.

The following sections use a small fixed-size 32-chain hash table (a larger 16,384-chain hash table is investigated in Appendix D on Page 346). This table is populated with 32 elements, one per hash chain. Sections 2.2.5 through 2.2.10 use a search-only workload.

Updates, as described above, are introduced in Section 2.2.11. In all cases, the experiment accesses and updates the hash table in a tight loop; there is no "think time" between accesses.

2.2.5 Code Locking

Perhaps the simplest way to protect the hash table is "code locking", using a global spinlock to protect the hash-table access functions. To use code locking, one would:

1. Acquire the global lock.
2. Search the hash table.
3. Release the global lock.

This approch has been used by a number of operating systems, for example, by the so-called "big kernel lock" (BKL) in the Linux kernel. The locking primitive used in this experiment is a simple test-and-set spinlock, implemented using cmpxchg instructions.

As can be seen in Figure 2.3, global locking scales negatively and does not offer ideal[2] performance even on a single CPU. The fact that AIX, DYNIX/ptx, HP-UX, Irix, Solaris, Tru64 Unix, and Microsoft Windows have provided good performance while scaling to at least several tens of CPUs provides a wealth of existence proofs that one can do much better.

Referring to the list of overheads in Section 2.2.3, this negative scaling from single-CPU to dual-CPU operation can only be due to lock contention or memory latency. Because the same code was executed in both cases, there should be no difference in instruction overhead or in pipeline-stall overhead. The next section therefore focuses on eliminating the lock contention in order to achieve non-negative scaling.

[2]In absence of hardware bottlenecks, "ideal" performance is computed by measuring the performance of the benchmark on a uniprocessor, but without use of any sort of synchronization mechanism, then multiplying by the number of CPUs.

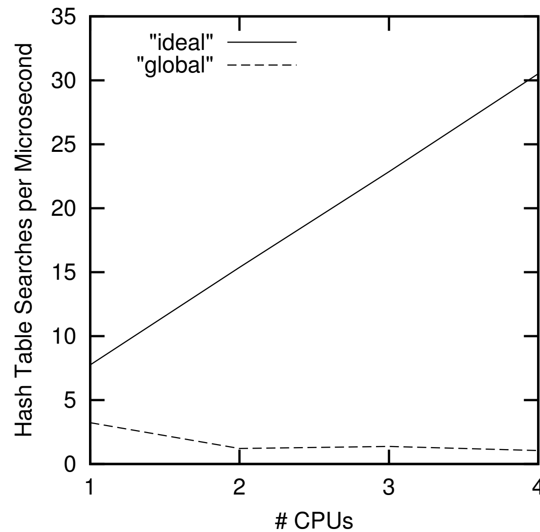


Figure 2.3: Globally Locked Read-Only Hash Table Performance

2.2.6 Reader-Writer Locking

Because the read-only variant of the hash-table mini-benchmark searches the hash table but does not update it, it is reasonable to ask whether it is possible to take advantage of the read-only semantics. A reader-writer lock [22, 85, 99] seems like a perfect fit for this situation. A reader-writer lock allows multiple readers to proceed in parallel, while writers must exclude both readers and other writers. Readers are still required to acquire the lock, but many readers may hold the lock simultaneously.

The use of reader-writer locking is similar to the global spinlock described in the preceding section:

1. Read-acquire the global lock.

2. Search the hash table.

3. Read-release the global lock.

This experiment uses a simple reader-writer lock that counts the number of readers and notes the presence of waiting writers, taken from the Linux kernel, but modified to use

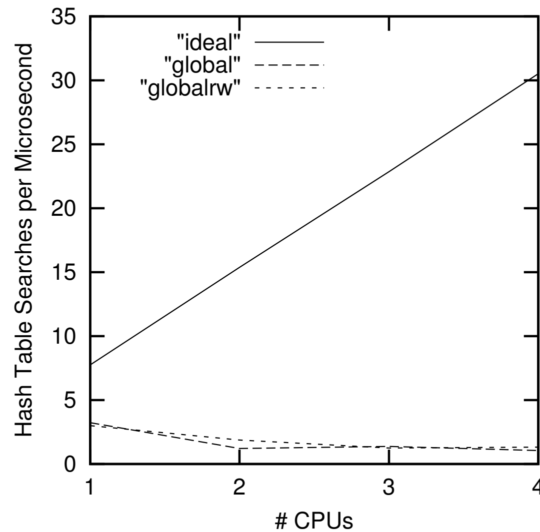


Figure 2.4: Globally Reader-Writer Locked Read-Only Hash Table Performance

cmpxchg in order to provide fair access to both readers and writers. As can be seen in Figure 2.4, although reader-writer locking does offer some improvement over the global spinlock, the effect is miniscule and the scaling still negative.

The cause of negative scaling cannot be lock contention, since this mini-benchmark performs only read-acquisitions of the lock. Just as with code locking, this cannot be due to instruction overhead or pipeline stalls, since the same sequence of instructions were used for both the uniprocessor and multiprocessor test cases. This leaves memory latency as the only possible cause of the negative scaling.

For example, on a four-CPU system, one would expect that 75% of the reader-writer lock acquisitions will find the data containing the reader-writer lock to be in some other CPU's cache, given this workload and this type of reader-writer lock. Referring again to Figure 2.1, it is clear that the read-side critical section must contain hundreds of instructions if it is to still be executing by the time the next CPU manages to read-acquire the lock. However, since a simple hash-table search can be accomplished in a very few instructions, the situation is as illustrated by Figure 2.5. In this figure, the vertical arrows represent time passing on two pairs of CPUs, one pair using reader-writer lock, and

the other using spinlock. The diagonal arrows represent data moving between the CPUs' caches. Note that the reader-side critical sections do not overlap at all – the overhead of moving the lock from one CPU to the other rivals that of the critical section, and, when combined with the memory-barrier overhead, exceeds that of the critical section. Hardware solutions to this problem are discussed in Section 2.2.15, but in the meantime, the focus is on currently available commodity hardware.

This example clearly demonstrates the high overhead of memory latencies in modern computer systems. The problem with the global reader-writer lock is that all the CPUs must manipulate a single data item, namely, the reader-writer lock itself, so that most of the CPUs' time is spent waiting for this single data item to be shuttled among their caches.

2.2.7 Partitioning and Data Locking

One way of ameliorating this effect would be for the CPUs to wait *in parallel* on the movement of multiple data items. One way to achieve memory parallelism is to increase the number of locks, assigning a separate lock to each hash chain. A CPU doing a search need only acquire the lock corresponding to the bucket to which the given search key hashes. Because CPUs tend to search different hash chains, they can incur memory latency in parallel. In contrast, the global reader-writer lock's memory latency was incurred serially.

Assigning a separate lock to a given data structure is termed "data locking", and is used heavily by highly scalable operating systems [9, 13, 29, 30, 31, 50, 67, 90, 104, 106, 107, 113]. Partitioning of data structures has also been used and researched heavily [16, 34, 82, 100, 134, 135, 137, 138].

This per-bucket locking approach requires larger changes to the hash-table search algorithm:

1. Hash the key.

2. Index into the hash array using the hash value to locate the corresponding hash bucket.

3. Acquire the lock associated with the hash bucket.

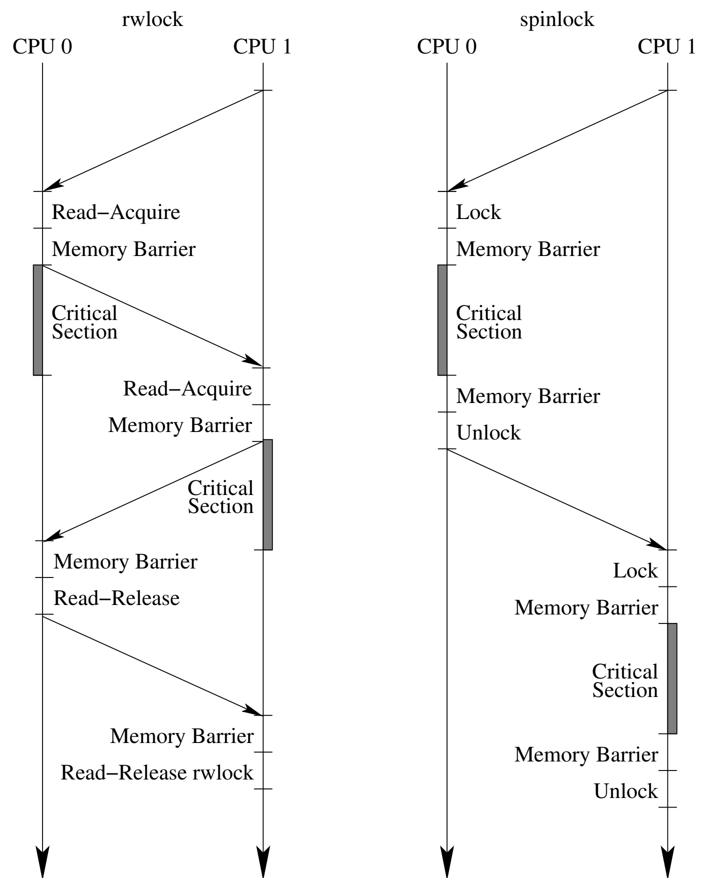


Figure 2.5: Comparing rwlock and spinlock

4. Search the list associated with the hash bucket.

5. Release the lock associated with the hash bucket.

Additional steps would be required to handle hash-table resizing.

As can be seen in Figure 2.6, this approach does achieve positive scaling, providing more than six times the performance of global locking on four CPUs. However, it still falls far short of ideal performance and scaling, as one would expect, given the SMMP-efficiency data shown in Figure 1.1 on Page 2.

If partitioning results in greater spatial or temporal locality, then performance and scaling may increase due to caching effects. However, in this experiment, the entire hash

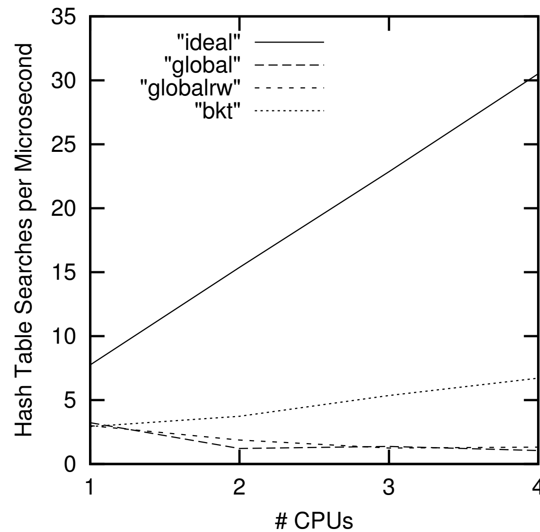


Figure 2.6: Bucket-Locked Read-Only Hash Table Performance

table fits in cache, and the access pattern is uniform and random, so no such increases ensue.

One way to reduce the synchronization overhead is simply to increase the size of the critical section, perhaps by increasing the length of the hash chains. However, in this case, this would simply cause the hash-table search to consume more CPU time, further *decreasing* performance. Increasing the size of the critical section can help in some situations [66], however, this is not one of them.

2.2.8 Combining Data Locking and Reader-Writer Locking

Taken separately, both global reader-writer locking and data locking performed better than did code locking. Perhaps the combination of the two would further improve performance.

Unfortunately this combination actually makes things worse, as can be seen in Figure 2.7. A close look at the single-CPU portion of this graph shows that global reader-writer locking is slower than global locking when running on a single CPU, which is to be expected given the greater cost of the reader-writer lock primitives, which must refer to the old value of the lock, which increases cost on this hardware, as will be seen in

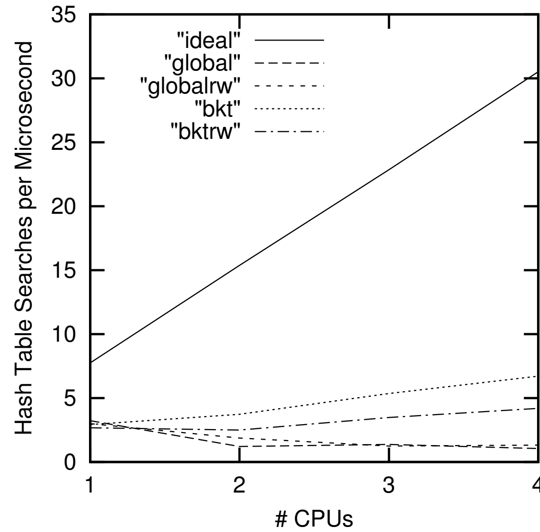


Figure 2.7: Bucket-Reader-Writer-Locked Read-Only Hash Table Performance

Section 2.2.10 on Page 29. When using global reader/writer locking on two CPUs, there is a certainty that both CPUs will be attempting to search the hash table concurrently, thereby benefitting from the ability of the reader-writer lock to permit multiple readers to proceed in parallel. In contrast, when there are 32 hash buckets, the probability of two CPUs accessing the same bucket at the same time is rather low, preventing the reader-writer lock from providing sufficient parallelism to overcome its higher overhead.

As is all too often the case, both in parallel programming and in life in general, combining two good things does not necessarily result in an even better thing.

2.2.9 Asymmetrical Reader-Writer Locking

The preceding two sections attempted to improve performance by incurring memory latency in parallel. It is only natural to ask whether one might be able to eliminate memory latency entirely. One possible avenue of attack is the scalable reader-writer lock demonstrated by Hsieh and Weihl [48], described by others [8, 120], analyzed by the author [68], and implemented in the Linux 2.4 operating system by Ingo Molnar as “brlock”.

The brlock primitive provides a separate lock for each CPU. A given CPU read-acquires

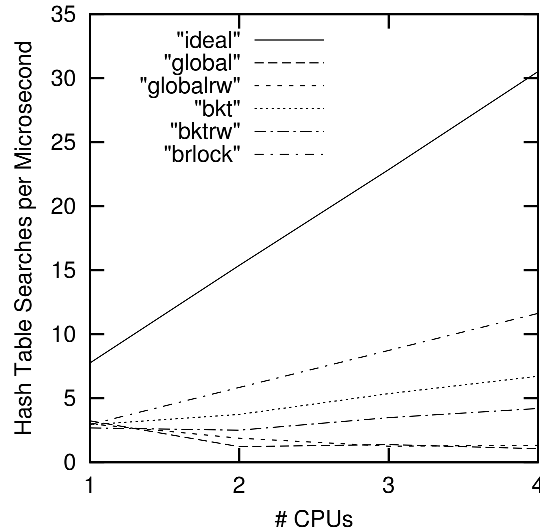


Figure 2.8: brlock Read-Only Hash Table Performance

the brlock by acquiring only its own lock, and write-acquires the brlock by acquiring all CPUs' locks. Thus, brlock provides excellent cache-locality for read-mostly workloads: in a read-only workload, reading CPUs will always find their lock in their own cache.

Note that brlock is an asymmetric primitive. Overhead is not eliminated; it is instead redistributed so that the read-side savings is realized at the expense of a large write-side penalty. The overall result may or may not be an improvement, depending on the workload. As can be seen in Figure 2.8, brlock does quite well on this read-only mini-benchmark, with linear scaling and almost twice the performance of the per-bucket lock when running on four CPUs.

2.2.10 Read-Side Performance Analysis

The single-CPU performance of brlock is still far from ideal. There has been substantial research into improving the performance of locking primitives at high levels of contention in SMMP systems [7, 35, 58, 60, 84, 85, 141], and in NUMA systems [52, 74, 95, 96, 119], but this is a simple SMMP system with low levels of lock contention. It is therefore necessary to look more deeply to understand brlock's performance shortfall.

The brlock primitive's degraded performance cannot be due to either lock contention or memory latencies, since brlock eliminates both sources of overhead entirely. This leaves instruction overhead and pipeline stalls as possible culprits. Because the number of additional instructions required to implement read-side brlock acquisition is quite small, it is reasonable to focus on pipeline stalls.

Indeed, it is reasonable to ask why pipeline stalls are required in the first place.

Pipeline Flushes Due to Memory Barriers

Modern CPUs tend to be superscalar, aggressively pipelined, with deep memory hierarchies. These features permit instruction execution and memory accesses to be performed out of order, constrained only by the requirement that the resulting execution have the same effect as sequential execution of these instructions *in the absence of any other changes such as those due to some other CPU.* Because synchronization primitives inherently involve multiple CPUs, instruction reordering must be suppressed, or at least constrained, in locking primitives. To see why, consider a pair of CPUs each executing the code in Figure 2.9.

```
1 spin_lock(&mylock);
2 tmp = a;
3 tmp = tmp + 1;
4 a = tmp;
5 spin_unlock(&mylock);
```

Figure 2.9: Locked Increment

In absence of instruction- and memory-ordering constraints, each CPU would be within its rights to execute this code in the order shown in Figure 2.10, which could cause increments to be lost.

```
2 tmp = a;
1 spin_lock(&mylock);
3 tmp = tmp + 1;
4 a = tmp;
5 spin_unlock(&mylock);
```

Figure 2.10: Out of Order Locked Increment

To prevent this from happening, spinlock primitives include memory barriers, as shown in Figure 2.11. Note that the exact sequence and type of instructions required to implement the memory barriers required by a spinlock varies, even within architectures. Here, the two `memory_barrier()` primitives prevent the memory references performed in the critical section (lines 8-10) from "bleeding out" from under the protection of the spinlock. However, most types of CPUs must stall their internal instruction pipelines and write buffers when executing a memory barrier, degrading performance.

```
 1 /* Begin spin_lock(&mylock); */
 2 while (test_and_set(&mylock)) {
 3   while (mylock == 0) continue;
 4 }
 5 memory_barrier();
 6 /* End spin_lock(&mylock); */
 7
 8 tmp = a;
 9 tmp = tmp + 1;
10 a = tmp;
11
12 /* Begin spin_unlock(&mylock); */
13 memory_barrier();
14 mylock = 0;
15 /* End spin_unlock(&mylock); */
```

Figure 2.11: Locked Increment With Memory Barriers

So pipeline stalls are the inevitable price of reliable locking primitives on many current pipelined and super-scalar CPUs. Hence, it is reasonable to ask how much these pipeline stalls cost.

Low-Level CPU Operation Costs

The preceding sections have demonstrated that the performance of SMMP software can be degraded by expensive low-level CPU operations such as data movement and pipeline stalls. Just how expensive are these operations?

Table 2.1 shows the measured costs of basic operations on a Sequent/IBM NUMA-Q system with four 700MHz P-III CPUs, which can retire two integer instructions per clock cycle. The atomic operation timings assume that the data already resides in the CPU's cache, and, since atomic operations implicitly include memory barriers on x86

CPUs, the 58.2 nanosecond cost of the atomic-increment operations includes memory-barrier overhead. Similarly, the 163.7 nanosecond cost of the CPU-local lock also includes memory-barrier overhead. This overhead accounts for much of the brlock single-CPU inefficiency seen in Figure 2.8. Note that all of these timings can vary, depending on the cache state, bus loading, and the exact sequence of operations. The data in this figure was collected using tight loops on a single CPU for all but the last two items, which were collected using tight loops on a pair of CPUs. In all cases, a fixed number of loops were executed, and the elapsed time measured. The variation among repeated measurements was quite low, lying within a few percent in all cases. These measurements are quite hardware dependent, however, the experiments described in this paper were run on a number of different CPU architectures, obtaining similar results [72].

Table 2.1: 700 MHz P-III Operation Costs

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Atomic Increment	58.2
cmpxchg Atomic Increment	107.3
Atomic Incr. Cache Transfer	113.2
Main Memory	162.4
CPU-Local Lock	163.7
cmpxchg Blind Cache Transfer	170.4
cmpxchg Cache Transfer and Invalidate	360.9

The atomic-increment cache-transfer measurement deserves some explanation, as one might naïvely expect that a cache transfer provoked by an atomic increment would cost roughly the same as one provoked by a compare-and-exchange (cmpxchg) instruction. The reason for the shortfall is that a given CPU may be able to execute several atomic increments before the other CPU manages to retrieve the cacheline containing the counter. Based on these numbers, each CPU is able to increment the counter twice before the other CPU retrieves the counter. This behavior will be extremely hardware dependent, and workload dependent as well. The experiments that measured the cmpxchg-induced cache

transfers took care to avoid this effect.

These overheads have been increasing relative to instruction-execution overhead. For example, on a 1.8 GHz P4, atomic increment costs about 75 nanoseconds, which is *slower* than the 700 MHz P3, despite the P4's having more than twice as fast a clock. Of course, projecting current trends is inherently risky, so an ideal synchronization technique would work well under all future scenarios.

It is reasonable to ask whether the performance attained by brlock is the best that can be expected, given the operation costs shown in Table 2.1. This question is taken up in the next section.

The Limits of Locking for Read-Only Searches

Figure 2.8 on Page 29 shows that a single CPU can execute roughly three brlock-guarded searches per microsecond, but can execute more than seven searches per microsecond in the absence of locking. The raw data shows that a brlock-protected search costs about 340 nanoseconds, while an unprotected search costs about 137 nanoseconds, a difference of 203 nanoseconds. From Table 2.1, a CPU-local lock costs about 164 nanoseconds, leaving only 39 nanoseconds unaccounted for.

This lesson is painfully clear. Insisting on the use of locking for read-only accesses more than doubles the cost of a simple hash-table search, and only about 10% of the total might be saved by clever coding techniques.

But read-only access is a special case. Although synchronization is not required among concurrent readers, it is still necessary among concurrent writers and between readers and writers. What happens when updates are performed concurrently with the read-only searches?

2.2.11 Write-Side Performance Experiment

The performance of a mixed search/update workload running on a 4-CPU 700MHz Pentium-III computer system is shown in Figure 2.12. As noted earlier, in order to simplify analysis, deletions and insertions are performed in pairs so that the number of elements in the hash table remains constant. The "ideal" performance is computed by quadrupling that of a

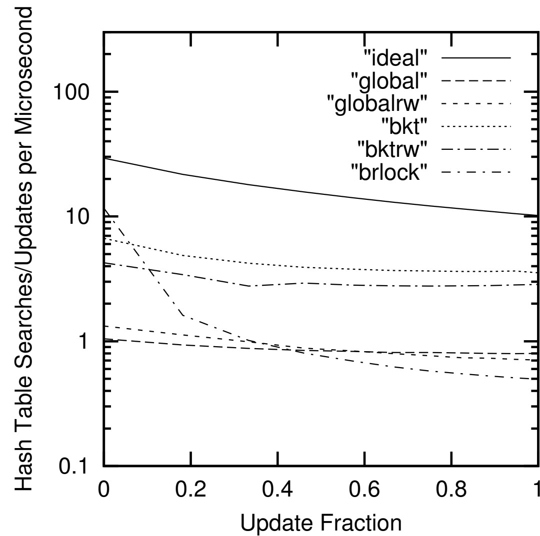


Figure 2.12: Hash Table Performance for Mixed Workload on Four CPUs

single-CPU implementation, which needs no locks, atomic operations, or synchronization of any sort.

The workload varies from read-only to update-only from left to right in this figure. Note that the y-axis is in logscale. The brlock primitive, which performs best for read-only workloads, quickly loses ground as the update fraction increases, performing worst of all for update-only workloads. Per-bucket lock emerges as the winner under these conditions. As with the read-only workload, it is natural to ask if this is the best that can be done.

Write-Side Performance Analysis

Per-bucket lock achieves about 3.5 updates per microsecond, compared to less than one update per microsecond for brlock. However, per-bucket lock's performance is still significantly worse than the roughly 10 updates per microsecond that would be achieved by an ideal technique that scaled perfectly from uniprocessor performance.

Because there are four CPUs, each CPU takes roughly 393 nanoseconds to do a perfectly scaling update, such as might be achieved when maintaining per-CPU data. Similarly, per-bucket lock requires about 1131 nanoseconds per update, for a difference of 783

nanoseconds, 164 nanoseconds of which are due to the overhead of the lock, including pipeline stalls due to memory barriers, leaving 574 nanoseconds.

Tracking down the source of the remaining 574 nanoseconds requires a more careful examination of the deletion and insertion operations. Deletion proceeds as follows, keeping in mind that the hash chains in this mini-benchmark each contain a single element:

1. Acquire the lock. Because some other CPU probably held the lock most recently, this CPU will incur a cache miss.

2. Pick up the pointer to the element from the hash bucket header. This incurs a cache miss.

3. Pick up the key from the element, which also incurs a cache miss. Assume that the comparison succeeds.

4. Deleting the element requires fetching its previous and next pointers, which are already cached due to the key comparison in the preceding step. Because there is but one element in the hash chain, both point to the hash bucket header.

5. The pointers in the hash bucket header are then updated to point to the header. Although the hash bucket header is already in this CPU's cache, it is in "shared" state, and so it must be invalidated from the other CPUs' caches.

6. Release the lock. Because this CPU just acquired the lock, the corresponding cache line most likely still resides in this CPU's cache.

7. Return the deleted element to the freelist. This entails adding the deleted element to a linked list, which requires invalidating the cacheline containing the element from the other CPUs' caches.

Thus, the deletion operation requires three cache misses and two invalidations. The insertion proceeds as follows:

1. Allocate an element from the freelist and initialize it. Because this CPU just freed this same element, the corresponding cache lines most likely still reside in this CPU's cache.

2. Acquire the lock. Because this CPU just acquired the lock, the corresponding cache line most likely still resides in this CPU's cache.

3. Update the new element's previous and next pointer to point to the hash bucket header.

4. Update the hash bucket header's previous and next pointers to point to the new element. Again, the cache lines corresponding to the hash bucket header most likely still reside in this CPU's cache.

5. Release the lock. Because this CPU just acquired the lock, the corresponding cache line most likely still resides in this CPU's cache.

The insertion normally does not incur any cache misses, so the pair of operations collectively incur the overhead of two lock round trips and three cache misses. Two of the three cache misses incur a later invalidation operation.

Next, the per-cache-line memory latency must be estimated. From Table 2.1 on Page 32, this overhead ranges from 170 nanoseconds in case of a blind write to the cache line to 361 nanoseonds in case of a read of the cache line followed by a write, which incurs the overhead of an invalidation as well as that of a cache miss. The lock acquisition's cache miss will be in the former category, but, as noted earlier, the linked-list manipulations fall into the latter category. Each deletion/insertion pair will therefore incur two memory latencies of 361 nanoseconds each and a third of 170 nanoseconds. Crediting each with half of the resulting 892 nanoseconds yields 446 nanoseconds per operation, which represents all but 128 nanoseconds of the 574 nanoseconds that was previously unaccounted for.

Because the total cost of each per-bucket-lock update was 1131 nanoseconds, there is only about 10% improvement to be had through optimizing at the instruction level.

However, it is fair to ask what would happen if the lock were placed in the same cacheline as the list header, as suggested for lightly contended locks by Gamsa et al. [29].

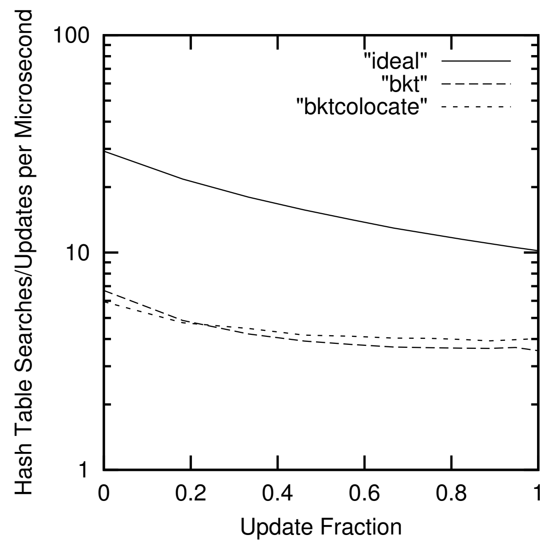


Figure 2.13: Hash Table Performance for Co-Located Lock

Experiment: Co-Located Locks for Updates

For updates, co-locating the lock in the same cache line as the hash bucket header would reduce the number of cache misses, but it would also mean that another CPU attempting to acquire this hash bucket's lock would remove the cache lines corresponding to the hash bucket header from this CPU's cache. Because there are 32 hash buckets and only four CPUs, updates should enjoy an overall performance increase.

However, since the entire hash table is small enough to fit into a CPU's cache, co-locating the lock does not reduce cache misses for searches, since searches would see cache misses only on the lock acquisition itself. Searches would therefore gain no benefit from the co-location, but would see the penalty due to other CPUs attempting to acquire the lock during a search operation. Searches would therefore bear the pain of an overall performance penalty.

Figure 2.13 shows the overall effect of this change. As expected, there is a modest improvement for updates and a modest penalty for searches. In either case, a key point is that reducing the number of cacheline transfers reduces memory-latency overhead.

Therefore, insisting on use of locking also sharply limits the update rate, even in the absence of significant lock contention. So the best locking primitive for read-mostly workloads is brlock, and the best for update-heavy workloads is the co-located per-bucket lock.

But is this the best one can hope for? One possible avenue for improvement can be seen in Figure 2.12 on Page 34. Here, the brlock trace decreases sharply with increasing update intensity, as expected given that write-acquisition requires acquisition of n locks, where n is the number of CPUs. As the number of CPUs increases, brlock's update performance will get steadily worse. This situation leads one to ask whether partitioning operating systems over groups of CPUs would be of benefit. This possibility is examined in the following section.

2.2.12 Operating-System Partitioning

A number of partitioned operating systems have been proposed, constructed, and evaluated, including the Hurricane project at the University of Toronto, the Cellular Disco project at Stanford University, and Larry McVoy's SMP Clusters proposal. The basic idea is to run multiple instances of the operating-system kernel, with the instance boundaries selected based on the underlying hardware. Presumably, these smaller operating-system instances would make more efficient use of the hardware than would a single monolithic operating-system instance, since they would not need to scale as high as would the monolithic instance. A software layer runs over the top of these multiple instances, presenting user applications with the illusion of a single monolithic operating system.

These projects and proposals are reviewed in the following sections.

Hurricane

The Hurricane operating system is described in a number of publications [134, 135, 137, 138]. This OS ran on "Hector", which was a 16-CPU NUMA system with 16MHz MC88100 CPUs. The number of CPUs making up a NUMA node could be configured. Each CPU had its own memory, so that the architecture of a single NUMA node in some ways resembled that of the more recent AMD Opteron™ [2].

Hurricane's design took the "demand-driven" [138] nature of operating-system kernels into account by driving the design with performance metrics that were tied to physical machine properties. The resulting analysis indicated that operating-system kernels must:

1. preserve the parallelism available in the application, for example, by not imposing unnecessary bottlenecks.
2. bound the overhead of servicing a given independent request.
3. preserve the locality inherent in the application.

This last item is especially important: the operating system must not act as a bottleneck for any embarrassingly parallel portions of the application.

Interestingly enough, the Hurricane researchers argued against page replication [135], perhaps because memory was still relatively expensive at that time. They also noted greater-than-desired lock complexity in Hurricane, and recommended following a strict locking hierarchy in order to simplify the locking and associated deadlock avoidance [134]. In this same paper, they introduce Tornado, and call out the fixed cluster size imposed by the Hector hardware as a major issue for Hurricane. Thus, Tornado was designed to run as a single OS image, but permitting each object or subsystem to cluster at whatever granularity is appropriate to its particular situation.

The Hurricane experience therefore cannot be taken as an unambiguous endorsement of the partitioned-operating-system approach.

Disco and Cellular Disco

Rosenblum et al. [100] produced an influential study of the effects of memory latency on operating systems and applications. Some time later, this same group produced Disco [16] and, later, Cellular Disco [34]. The underlying assumption seems to have been that commodity operating systems would not scale, thus they focus on a layer of software that permitted multiple copies of such an OS to run on a single large computer system. Although the Cellular Disco approach is quite prominent today in products such as VMWare,™ commodity OSes such as Windows™ have since scaled to world-class levels on 64 CPUs, as measured by the TPC/C benchmark on `http://www.tpc.org`.

SMP Clusters

McVoy recently proposed implementing a single-system image operating system composed of multiple cooperating operating-system instances running on a single SMMP computer system [82]. The idea is that the individual instances (OSlets) would remain relatively simple, due to the smaller number of CPUs that each would be responsible for. This proposal has some interesting properties, but also some severe drawbacks, particularly those relating to races and corner cases involving inter-OSlet communication and I/O interrupts [76]. If this system is implemented, it will be interesting to compare it to its predessors.

Partitioning Discussion

The partitioning of OS kernels was intended to increase scalability, so that individual kernels each scaling to four CPUs could be combined into a larger system that scaled to many tens of CPUs. However, AIX, DYNIX/ptx, HP-UX, Irix, Solaris, Tru64 Unix, and Windows already scale to at least several tens of CPUs, as evidenced by benchmark results on `http://www.spec.org` and the aforementioned `http://www.tpc.org`. In addition, existing partitioned OSes did not provide the expected simplification, exemplified by the Hurricane researchers taking a different approach with Tornado, partitioning individual OS subsystems as appropriate for the component in question, rather than fitting the entire kernel to a hardware-centric Procrustean bed [134]. Further, a number of workers have demonstrated that several important algorithms can be parallelized, including computer-communications protocol processing [31, 90, 104], general-purpose kernel memory allocation [75, 80], allocation of predefined data structures [14, 15], and timed-event processing [77]. Finally, there is some reason to believe that at least some of the complexity that has been attributed to scalability is due to "work hardening" of code resulting from the incremental approach to scalability that has been taken in many operating-systems projects [83].

In addition, partitioning does not completely solve the scaling problem. There remains a need to build a global view of the machine, and this global view requires some form of

coordination, consensus, and synchronization.

Because blindly partitioning the entire operating system on hardware boundaries is inflexible, further investigation into monolithic implementations is warranted. However, experience with locking indicates that it results in excessive complexity, severe performance and scaling limitations, or both. Can locking, and thus these disadvantages of locking, be eliminated?

2.2.13 Simple Non-Blocking Synchronization

Non-blocking synchronization (NBS) is defined to be any linearizable[3] synchronization mechanism where each thread in the system is guaranteed to complete an operation after taking a finite number of steps in the absence of interference by other threads [44]. Mechanisms built on the simple spinlock described in Section 2.2.1 cannot possibly be examples of non-blocking synchronization, because in principle, an extremely unlucky thread might spin forever if other more fortunate threads were able to monopolize the lock.

In many cases, it is possible to avoid the use of locks. For example, instead of using locks to protect a counter, one could use the lighter-weight atomic increment instruction provided by the Pentium CPU, thereby meeting the definition of NBS. However, this instruction incurs substantial pipeline-stall overhead as shown in Table 2.1, so that an atomic increment, though considerably cheaper than a lock, is two orders of magnitude more expensive than a simple increment instruction. The situation only gets worse with the addition of more CPUs, since the even larger memory-latency overhead will then be incurred.

The pipeline-stall and memory-latency overheads are required in order to satisfy the strict semantics of atomic increment, which dictate that a counter take on a well-defined sequence of values that is visible to all CPUs. Although these strict semantics ensure correctness in almost all cases, there are situations in which less-strict semantics can provide a correct solution at much lower cost. The basic problem is that strict semantics require frequent communication among the CPUs, communication that is extremely expensive.

[3] A "linearizable" synchronization mechanism is one in which all CPUs observe the same sequence of values for all corresponding data structures.

This raises the question of whether relaxed semantics may be used, reducing the required amount of communication.

One possible way to relax the increment operation's semantics is to note that increment is commutative, so that the result of a series of increments will be the same regardless of the order in which they execute. One way to take advantage of this communitivity is for the increment operation is to use a separate cacheline-aligned variable for each CPU. CPUs increment this "split" counter by incrementing their own variable, which yields perfect cache locality, so that the increment operation never needs to access data in some other CPU's cache, thereby greatly reducing the cost of each increment.

However, to read out the value of the counter, a given CPU must sum all of the CPUs' values, so that reducing the cost of the increment operation has increased the cost of the read-out operation. This implementation relies on the commutative law of addition: in summing up the per-CPU variables, all the increments from the first CPU are performed first, followed by those of the second CPU, and so on. Because cacheline-aligned loads and stores are atomic, there is no possibility of accessing a half-incremented variable, and since all operations are unconditional, this procedure qualifies as NBS, but only if all operations increment by a fixed value.[4]

As long as the value of the split counter is never read out, the cost of an increment decreases by more than an order of magnitude compared to that of an atomic increment of a global variable. However, the cost of reading out the value will be quite high, and will rise with the number of CPUs. Furthermore, reading out the value on one CPU will cause each of the other CPUs to incur a large penalty the next time that they increment the split counter, as shown in Figures 2.14 through 2.18. Figure 2.14 shows the initial cache configuration. The split counter is made up of the two variables A (for CPU 0) and B (for CPU 1). Neither of these variables is present in either of the CPUs' caches.

When both CPUs increment the split counter, they must load their respective variables from memory into their caches, as shown in Figure 2.15. Note that although a copy of each

---

[4]If multiple increment values are permitted, the resulting execution sequence will not be linearizable. For example, given a counter initially at zero, if CPU 0 increments by 1 concurrently with CPU 1 incrementing by 2, CPU 2 might observe the sequence 0,1,3 while CPU 3 observes the sequence 0,2,3, violating the linearizability requirement.

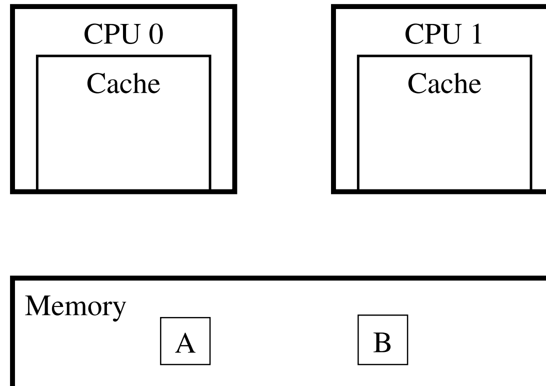


Figure 2.14: Split Counter Initial Cache Configuration

variable still resides in memory, these copies are outdated, as indicated by the shading. The up-to-date values of the variables are found only in the CPUs' caches. Note that loading the values from memory is quite expensive, costing around two orders of magnitude more than loading values from cache.

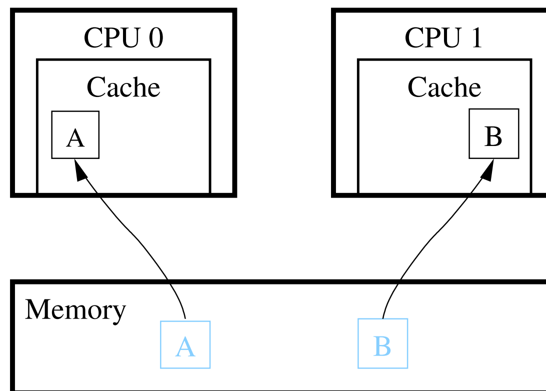


Figure 2.15: Split Counter First Increment

However, subsequent increments find the variables already in the CPUs' caches, as shown in Figure 2.16, and therefore run at full speed.

Suppose that CPU 0 reads out the counter. It does this by summing variables A and B. Variable A already resides in CPU 0's cache, but it must fetch the value of variable B from

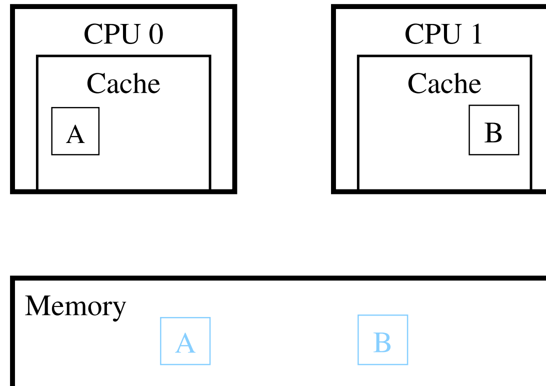


Figure 2.16: Split Counter Subsequent Increments

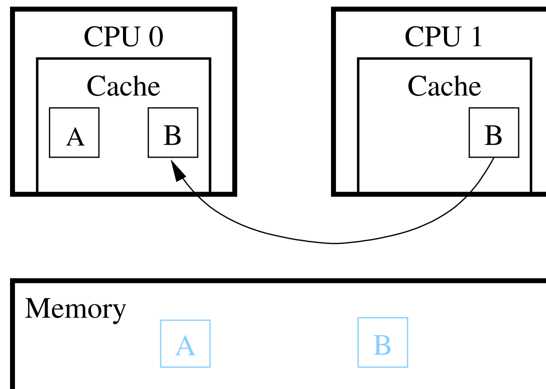


Figure 2.17: Split Counter Readout

CPU 1's cache, as shown in Figure 2.17. This operation is less expensive than fetching from memory, but still orders of magnitude more expensive than if variable B already resided in CPU 0's cache.

Note that since CPU 0 is only reading variable B, both CPUs now have a copy of it in their caches. This means that neither CPU may change variable B, since doing so would result in inconsistent values for this variable in the two CPUs' caches. Therefore, the next time that CPU 1 wishes to increment the split counter, it must cause CPU 0 to invalidate its copy of variable B, as shown in Figure 2.18. After this invalidation operation completes, the situation will again be as shown in Figure 2.16, so that increments of the split counter

may proceed at full speed on both CPUs. By avoiding expensive atomic operations, split counters can be incremented extremely efficiently, and read out more slowly. They are therefore well-suited to maintaining statistics, such as the TCP/IP packet counts found in many operating-system kernels. In this situation, the split counter will be incremented frequently, namely, on every packet, but read out infrequently, namely, only when the system administrator is investigating some sort of networking problem.

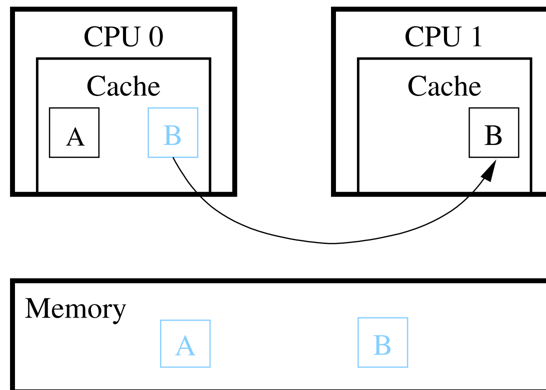


Figure 2.18: Split Counter Increment After Readout

This split-counter approach works well for statistical counters because it is *asymmetric*, favoring the frequent increment operations and imposing more work on the rare read-out operations. This asymmetry permits the implementation to bind data to the CPUs that are using it more frequently, promoting locality and greatly reducing cache thrashing. In addition, this implementation eliminates special atomic instructions and their associated pipeline stalls, and is immune to contention.

Although these statistical counters are very specialized, similarly specialized simple non-blocking constructs are used heavily in operating system kernels. This raises the question of whether non-blocking synchronization may be generalized so as to eliminate pipeline-stall, memory-latency, and contention overheads entirely.

2.2.14 General Non-Blocking Synchronization

It makes sense to first focus one's attention on a single source of overhead, and of the four sources of overhead called out in Section 2.2.3, contention has received the most attention. This is quite understandable for work done before 1990, since, as can be seen in Figure 2.1, the overhead due to memory latency was then quite manageable. In addition, CPU pipelines were quite short, so that pipeline-stall overheads were also quite small. Therefore, before the 1990s, it made good sense to focus solely on eliminating lock contention.

One way to eliminate lock contention is to make use of atomic operations, which themselves are based on the atomic-instruction facilities that are present in almost all modern microprocessors. Workers have made use of such facilities for several decades, as witnessed by the CDC 3300's "set destructive load" instruction [20] from the mid-1960s. However, work in this area was at best a collection of ad-hoc algorithms specialized to the atomic-instruction facilities of a particular platform.

Herlihy changed this situation by introducing non-blocking synchronization [42], which placed the use of such atomic-instruction facilities on a firm theoretical foundation and allowed these facilities to be generally applied. NBS is a set of algorithms that use low-level atomic instructions, such as compare-and-swap or load-linked and store-conditional, to make data-structure updates and traversals appear to be atomic.

For example, a conceptually simple NBS algorithm might update a dynamically allocated data structure by making a copy of it, updating the copy, then atomically updating the pointer to point to the new copy, but only if the pointer is unchanged. If two threads attempted concurrent updates, one would succeed, but the other would fail upon finding that the pointer had been changed by the successful thread. The failing thread would then discard its copy and retry. This thread is shown in detail in Figure 2.19, where thread 1 is attempting to add element D to and thread 2 is attempting to delete element B from a list initially containing elements A, B, C, and D. In step (2), both threads have created copies of the initial list and performed their updates. In step (3), thread 1 wins the race, succeeding in atomically updating the pointer so that it points to thread 1's list instead of

the initial list. Process 2 fails to atomically update the pointer, since it no longer points to the intial list. In step (4), thread 2 retries its deletion of element B, copying the new list and performing the deletion from the copy. Note that the original list has been freed up, as it is no longer used. This operation is glossed over, since safely freeing NBS data structures is quite complex, and is beyond the scope of this discussion; interested readers are referred to the extensive NBS literature [42, 87]. In step (5), thread 2 has succeeded in atomically updating the pointer from the list that thread 1 created to thread 2's new list.

The atomic nature of NBS entirely eliminates lock contention. Unfortunately, NBS typically introduces other forms of contention such as repeated high-overhead retries, as seen in this example. Furthermore, NBS uses the same low-level instructions that are used to implement locking primitives, and therefore suffers from the same high memory-latency and pipeline-stall overheads as does locking on modern SMMP systems. It is nevertheless worthwhile to review NBS, because many of the ideas developed in the course of refining NBS are valuable in their own right.

Herlihy's work distinguishes among several variants of NBS:

1. non-blocking synchronization: each thread in the system is guaranteed to complete an operation after taking a finite number of steps.

2. lock-free synchronization: at least one thread in the system is guaranteed to complete an operation after the system takes a finite number of steps.

3. obstruction-free synchronization: each thread in the system is guaranteed to complete an operation after taking a finite number of steps in the absence of interference from other threads [44].

All three variants of NBS permit safe operation in the presence of preemption and thread death, and have seen much subsequent activity. NBS-based algorithms carefully manipulate pointers and corresponding version numbers such that halting any one thread at any point would still permit all other threads to proceed.

For a simple example, consider a non-blocking push onto a stack, as shown in Figure 2.20. Here, the new element is linked to the current stack head on line 11, and the

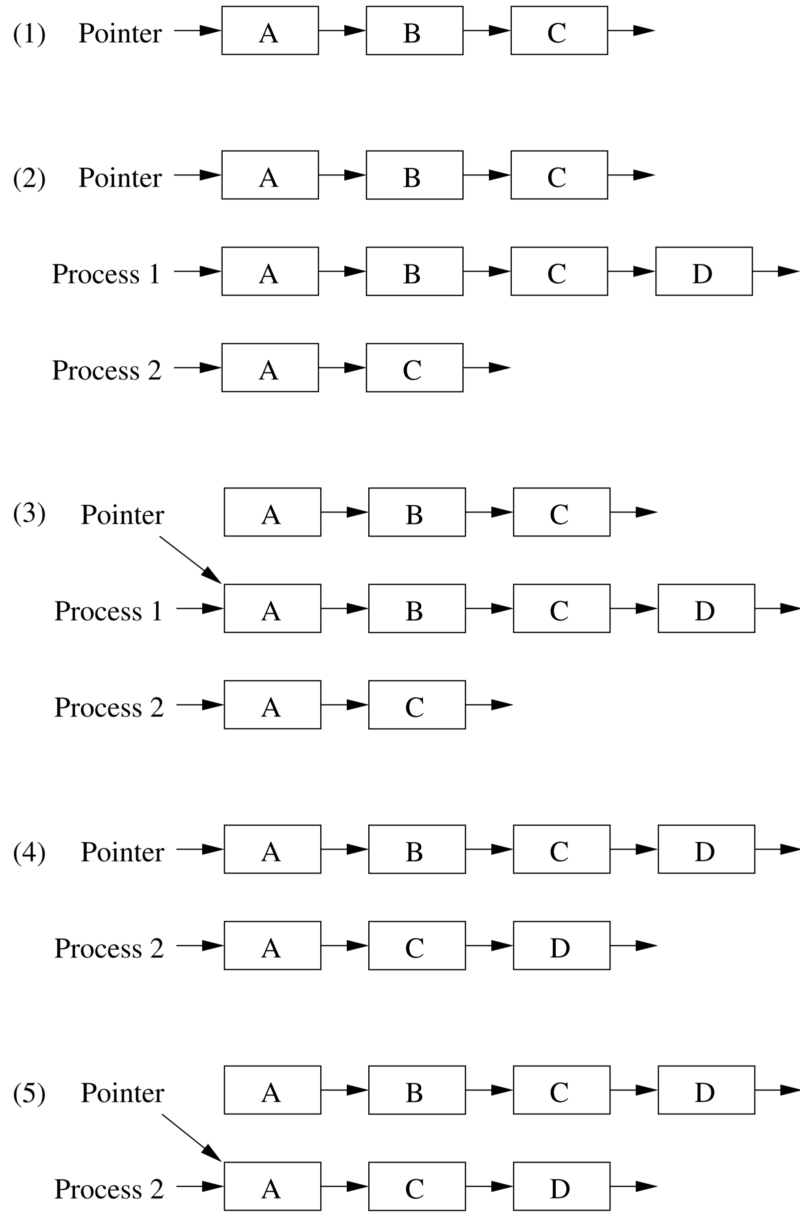


Figure 2.19: NBS Update: Conceptual View

head pointer is atomically updated on line 12. The `cmpxchg()` primitive takes a pointer to the value to be updated as its first argument, the presumed old value as its second argument, and the new value as its third argument. The loop spanning lines 10-12 repeats until the update succeeds.

Interference due to other CPUs manipulating this same stack can cause repeated failures and retries, with each such retry requiring another pass through the loop. This failure-retry behavior brings out a key point about NBS, namely, that it suffers from contention effects, but these effects take the form of repeated retries rather than spinning. Because each NBS retry is attempting to change the state of the system, high levels of contention can exact a severe performance penalty due to the resulting load on the computer's memory system. That said, in many cases, NBS naturally restricts the scope of contention of a pair of concurrent updates to the intersection of the sets of data items affected by the updates. In addition, as with locking, backoff techniques may be used to reduce the overhead of contention.

```
 1 struct el {
 2   struct el *next;
 3   int data;
 4 };
 5 struct el *head;
 6
 7 void
 8 push(struct el **head, struct el *p)
 9 {
10   do {
11     p->next = *head;
12   } while (cmpxchg(head, p->next, p) != p->next);
13 }
```

Figure 2.20: Non-Blocking Push

Any thread that dies suddenly will do so either before or after the atomic compare-and-exchange operation on line 12 completes successfully. If it dies before, then the element has not been pushed into the stack. On the other hand, if it dies after, the element will have been pushed into the stack. In either case, the stack will be correctly formatted, and other threads may proceed.

However, constructing the corresponding pop operation is significantly more challenging. It is quite tempting to try the implementation shown in Figure 2.21, but this implementation is fatally, though subtly, flawed [133].

```
 1 struct el *
 2 pop(struct el **head)
 3 {
 4   struct el *p;
 5   struct el *q;
 6
 7   do {
 8     p = *head;
 9     q = p->next; /* BUG!!! */
10   } while (cmpxchg(head, p, q) != p);
11 }
```

Figure 2.21: Non-Blocking Pop

To see this, consider a stack containing elements A, B, and C, in that order from the top of the stack. Suppose that CPU 0 starts a pop operation, completing execution of line 9, then being delayed. CPU 0's variable `p` will then be pointing to element A and its `q` variable will be pointing to element B. Suppose that CPU 1 now executes two pop operations, removing elements A and B from the stack, leaving only element C. Suppose further that CPU 1 then pushes element A back onto the stack, so that the stack now contains elements A and C.

Then CPU 0 will execute the compare-and-exchange operation on line 10, which will find that `head` is still pointing to element A, and will therefore succeed–but will cause `head` to point to element B, which is no longer on the stack! This ABA problem is due to the that compare-and-exchange cannot detect a sequence of changes if that sequence results in the same state, in this case, the pointer again pointing to element A. Successful use of non-blocking and NBS requires careful consideration of complex scenarios such as the one just described. The fact that this complex and subtle example arose in an extremely simple data structure should give pause to anyone considering use of these techniques on larger and more complex data structures.

Although use of load-linked/store-conditional in place of compare-and-exchange can simplify matters somewhat [42], in general, NBS imposes expensive atomic operations to

shared memory on readers, requires "fat pointers" augmented with counters to check for structure reuse, requires expensive copy operations, requires type-safe memory, and possesses efficient implementations for a relatively small number of parallel algorithms. In addition, there are no known general techniques to permit a large data structure to be incrementally converted from locking to non-blocking synchronization. Any such conversion may therefore need to be made in a "big bang" fashion, at one go. This last point is quite important for software-engineering reasons, since it means that any change from locking to NBS will be a relatively large, high-risk change, thus inhibiting NBS adoption.

Non-Blocking Synchronization Data Element Reuse

Much of NBS's complexity is due to the fact that threads can be delayed for long time periods due to interrupts, preemption, and repair of "soft" hardware errors such as single-bit ECC errors in memory transactions. Such potential delays limit NBS's ability to reuse memory corresponding to data elements removed from a data structure, since some other thread might still be referring to it. As noted earlier, NBS has used "fat pointers" that include counters that are incremented on each update, so that readers can determine that they are referring to an obsolete data element. However, for the readers to make this determination reliably, the data structure's counters must remain at the same offset within the structure after reuse. To meet this same-offset requirement reliably, NBS depends on "type-safe memory", in which memory, once used for a given type of data structure, may never be used for any other data structure type.

This type-safe-memory constraint leaves systems using NBS vulnerable to denial-of-service attacks, where the attacker overloads the system with a specific type of transaction that causes most of memory to be dedicated to the corresponding type of data structure. The system is then unable to respond to overloads due to other transaction types, since it is unable to re-purpose its memory.

Therefore, Michael [86, 87] has proposed deferred-destruction techniques that eliminate the requirement for reuse checks and type-safe memory by requiring that each thread traversing a NBS-protected data structure keep a separate list of "hazard pointers" recording which data elements that the thread is still referencing. Any data element removed

from a list may be repurposed as soon as all of the hazard lists are free of pointers to this element. Herlihy et al. [43, 45] propose a similar scheme. However, these techniques imposes significant performance penalties [27], as would be expected given the need to for read-only accesses to write to shared memory. Nonetheless, deferral of destruction is an important concept with a long history [56]; the discussion in Sections 2.2.19 and Section 2.2.20 show how it may be exploited.

Herlihy suggests "freezing" a block while reading it in order to avoid that block being freed and reallocated while being read [40], but notes that doing so greatly complicates the protocol [42]. Another way to avoid such inopportune recycling is to combine NBS with RCU, as was done for the K42 operating system's hash tables, described in Section 6.7 on Page 221. This combination of RCU with NBS appears to address many of the issues noted above.

Anderson and Moir [5] have proposed techniques for large-structure updates, but these techniques still require that readers write to shared storage, with disastrous effects on memory locality.

Obstruction-Free Synchronization

Herlihy, Luchangco, and Moir [44] recently introduced "obstruction-free synchronization", which only guarantees forward progress in absence of contention. This is a weaker guarantee than lock-free synchronization (which guarantees that some thread will make forward progress in a finite number of timesteps) and non-blocking synchronization (which guarantees that all threads will make forward progress in a finite number of timesteps). Obstruction-free synchronization promises simpler and more efficient algorithms than either lock-free or non-blocking synchronization, due to the fact that the NBS code that guarantees forward progress may be eliminated. However, it still requires writes to shared storage during read-only accesses.

Software Transactional Memory

More recently, Keir Fraser [27] extended the notions of software implementation of multiple compare-and-swap (MCAS) and software transactional memory (STM). A key concept in

this work is the maintenance of multiple versions of each data item, so that a given CPU can safely carry out its operation on its version of the data structure, even if that operation is invalidated by concurrent activity of some other CPU. Fraser was able to show performance rivaling that of lock-protected data structures, but only tested data structures with natural bottleneck points, namely, skip lists [93, 94] and binary search trees. In addition, Fraser's algorithms require an additional level of indirection, doubling the number of cache lines that must be fetched, and also require expensive writes to shared memory, even for read-only accesses. Researchers at Sun [45] have worked along similar lines.

Synthesis

Massalin and Pu [92, 65] describe Synthesis, which is an operating-system kernel using non-blocking techniques. Note that Synthesis did not pervasively use non-blocking techniques, and that although some non-blocking and lock-free techniques have been introduced into mainstream operating systems, as exemplified by the non-blocking hash table described in Section 6.7 on Page 221, there has not been widespread use of kernels based entirely on these techniques.

Cache Kernel

Cheriton implemented a kernel based entirely on non-blocking synchronization [18], and later, with Greenwald, published an analysis of this experience [36]. Greenwald and Cheriton argue that non-blocking synchronization provides the following benefits:

1. Reduction of self-deadlock concerns.

2. Avoidance of priority-inversion issues.

3. Greater insulation from fail-stop failures.

The first benefit is real, but solutions exist for lock-based operating-system kernels. For example, the Linux kernel provides primitives for simultaneously acquiring spinlocks and disabling interrupts, such as the `spin_lock_irqsave()` primitive.

The second benefit is also real, but greatly ameliorated by the engineering design practice of keeping contention low, and through use of "priority inheritance" techniques that raise the priority of the holder to at least that of the most important waiting thread. Locking primitives that suppress preemption, such as those in the Linux 2.6 kernel, are also helpful in this regard.

The third benefit seems more applicable to user code. Kernel threads can fail in diverse ways, many of which are anything but fail-stop. Cheriton does not present any evidence indicating that there are sufficient fail-stop bugs in kernel code to warrant the complexity of non-blocking synchronization.

Cheriton's non-blocking primitives require that readers write to shared memory in order to verify that the data structure was unchanged during the read. Given the high cost of writes to shared memory, one would not expect large performance benefits from these primitives under conditions of low contention. And in fact, Cheriton's results show that these non-blocking primitives have little or no advantage over spinlocks until the load rises to the point that threads holding locks have a high probability of being preempted. As noted earlier, locking primitives that suppress preemption are a viable alternative in operating-system kernels.

Cheriton's primitives also require a double-compare-and-swap (DCAS) instruction that takes two addresses. This instruction is not available on most popular microprocessors. It may be simulated in software, but for this to work, readers must either take locks or be coded to tolerate non-atomic simulated DCAS instructions.

In addition, Cheriton's primitives require use of type-safe memory, which prevents re-purposing memory. Such re-purposing is required if a kernel is to support a dynamically changing workload.

It is also quite interesting to note that Cheriton does not claim performance benefits or significant complexity benefits.

Nonetheless, NBS techniques are widely used for specific cases where simple and efficient implementations exist. However, the general-purpose transformations are not heavily used, in part due to lack of an efficient implementation. But the implementations are limited in part by the capabilities of the hardware. Could more capable hardware be brought

to bear on synchronization problems?

2.2.15 Transactional Hardware

Various forms of transactional hardware has received much research attention over the past few years. Representative examples that are compatible with existing SMMP locking software include transactional lock removal (TLR) [98] (which is based on earlier work with speculative lock elision (SLE) [97]) and thread-level speculation (TLS) [64] (which is based on earlier work with speculative locks [63]).

TLR hardware recognizes critical sections based on standard locking primitives, thus requiring no change in software. The hardware executes the entire critical section as one atomic operation, using timestamping and rollback to resolve conflicting accesses, and varying priorities to avoid livelock. Note that hardware rollback is reasonable, as all CPUs that do speculative execution are capable of such rollback.

However, because TLR uses the CPU cache to record tentative state prior to atomic commit, the CPU's ability to atomically execute a critical section will be limited by its cache capacity and geometry. Although there are well-known technologies, such as victim caches, that can ameliorate these limitations, there will always be limits. Such limits are data-layout-dependent, and can therefore result in unpredictable slowdowns in those cases where the CPU caches cannot accommodate the critical section's data. Such slowdowns become increasingly unpredictable with the advent of multithreaded CPUs that share cache resources.

TLS also leverages the hardware work on speculative execution to enable atomic execution of critical sections. TLS uses a "safe thread" that always executes non-speculatively in order to guarantee forward progress. Since the safe thread executes non-speculatively, it actually acquires and releases the locks. Note that speculative threads may execute critical sections that would normally be excluded as long as there are no data conflicts, however, this also means that the speculative threads cannot commit their state until after the safe thread releases the lock. Again, this means that the ability to speculatively execute critical sections depends on the size and geometry of the CPU cache.

Such transactional hardware does look promising, but it is still a topic of research, and

will therefore likely take many years before it is available in commodity CPUs. The fact that such hardware does not require any changes to software (not even recompilation) is quite positive compared to earlier approaches that did require such changes [46]. However, given that speculation increases power consumption, which is becoming a limiting factor in many current designs, it is possible that transactional hardware will face significant barriers to widespread adoption.

In the meantime, we must make due with available hardware, and therefore must still face the fact that general-purpose NBS algorithms lack efficient implementations. But perhaps this very generality is actually part of the problem. Is it possible to improve performance by relying on the specifics of the problem at hand? Can semantics be exploited to provide fully parallel implementations for more general classes of operations?

2.2.16 Exploiting Semantics

Weihl and Liskov [140] advocate taking advantage of specific properties of the underlying data and operations on that data, such as commutivity, in order to increase parallelism. This is good advice, and in fact was followed in the construction of the split counter described in Section 2.2.13 on Page 41 but helps only in very specific situations.

However, in many cases, the order of non-commutative operations is left unspecified. For example, consider creation and deletion of a pre-existing System V IPC object. If the creation precedes the deletion, the creation will fail, and the object will be deleted. If, on the other hand, deletion precedes creation, both operations will succeed, and a new object will be created to replace the old one. But if the creation and deletion operations are being carried out by independent user threads, the kernel is permitted to arrive at either result. The higher-level semantics of such operations state that when there is no causal relationship between two events, they may proceed in either order.

This System V IPC case permits reads to proceed in parallel, in contrast to the split counter, where writes proceed in parallel. It is therefore critically important to select the semantics to be exploited carefully so as to give maximum benefit to the common-case operations. How can operation reordering be exploited in common operating-system algorithms, such as those involving linked lists?

2.2.17 Read-Mostly Linked-List Insertion

To answer this question, first consider inserting an element into a linked list on a computer system with a sequentially consistent memory model.[5] If the readers never acquire locks, then the insertion must be done atomically, so that a given reader either sees the list before the insertion, or sees the list with a fully-fledged new element. Updates must still synchronize with each other, but, in order to obtain good performance, readers must not be required to use the expensive operations required to synchronize with updaters. Figure 2.22 shows a sequence of steps that accomplish this atomic insertion:

1. The initial list contains elements A and C.
2. Element B is allocated and initialized, including its pointer to element C.
3. Element A's successor pointer is updated to point to element B. Note that a reader currently accessing element C might or might not have passed through element B, depending on the timing, as indicated by the shaded arrows emanating from element A.
4. Any subsequent traversal of this list is guaranteed to see element B.

The key point here is that step 3 of this sequence makes a complex series of updates visible atomically–no CPU will see element B half-initialized. This example illustrates the usefulness of hiding complex operations behind carefully chosen "commit points" in order to eliminate the need for the readers to synchronize with updaters.

2.2.18 Read-Mostly Linked-List Removal

The insertion algorithm works well, and has been used for decades. However, sooner or later it will be necessary to remove an element from this list. Again, if the readers do

[5] Sequential consistency guarantees that all CPUs agree on the order of all writes to memory. Computer systems with weaker memory models (almost all of them!) must execute "memory barrier" instructions to force ordering when necessary. For example, locking primitives contain memory-barrier instructions in order to prevent operations making up the critical section from "bleeding out" into the surrounding code, as discussed in Section 2.2.10 on Page 30.

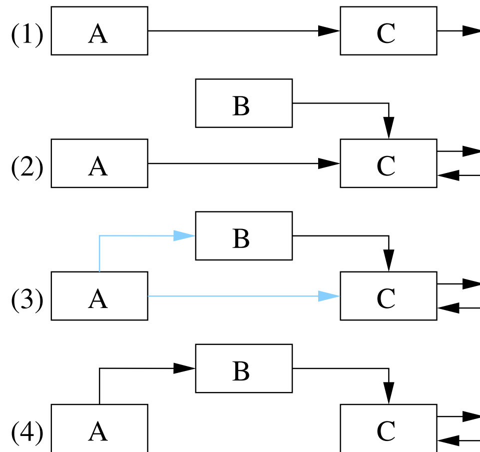


Figure 2.22: Atomic Insertion Into a Linked List

not use locking, removal must be done atomically. This may be accomplished as shown in Figure 2.23:

1. Initially, the list contains elements A, B, and C in that order.

2. Element A's successor pointer is updated to point to Element C. At this point, a reader referencing element C may or may not have passed through element B, as indicated by the shaded arrows emanating from element A.

3. Any new traversal of the list is guaranteed not to see element B.

This algorithm can produce inaccurate results–readers can see stale data, and the order of the readers' list traversals are now unpredictable (as they would also be with reader-writer locking). For example, a reader might be examining element B after it has been removed from the list. However, it turns out that there are a number of situations where such stale data can be tolerated, such as TCP/IP routing tables. Because TCP/IP routing protocols have built-in delays to prevent route thrashing, a given computer's routing tables may have stale data in any case. The TCP/IP protocol compensates for this by retransmitting any packets that are lost due to being sent down the wrong path. Because the routing-protocol delays are measured in seconds or even minutes, the additional sub-second stale-data delays are insignificant.

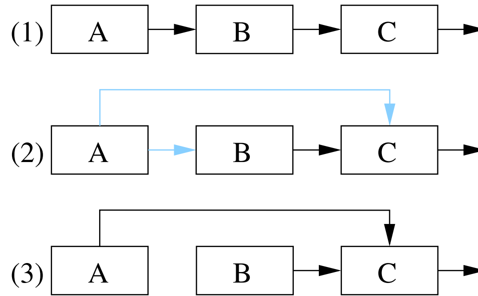


Figure 2.23: Atomic Removal From a Linked List

2.2.19 Read-Mostly Linked-List Removal and Reclaiming

Again, this algorithm works quite well, and is another illustration of the value of hiding complex operations behind carefully chosen commit points. However, unless wantonly leaking memory is permissible, it begs the question as to when element B may safely be reclaimed by returning it to the free pool. Of course, since element B has been unlinked from the list, there is no way to acquire a new reference to it. But since the readers are in no way making their presence known, and since a given reader with a pre-existing reference to element B might be indefinitely delayed by interrupts, correctable ECC errors in memory, and so on, how can one guarantee that all such pre-existing references have been released?

Normally, such a guarantee is based on direct observation of state manipulated by the readers, reader-writer locks being a typical example. However, in this case, the readers manipulate no state, for so doing would require use of expensive synchronization operations. Therefore, any such guarantee would have to be based on some more indirect means. A key question is whether any such indirect means exists.

It turns out that in some environments, including operating-system kernels, server applications, and event-driven systems, it is in fact possible to determine indirectly when all such pre-existing references have been released. For concreteness, consider a non-preemptive operating-system kernel. In this environment, it is illegal to block while holding a pure spinlock, since doing so can result in a deadlock situation where the thread holding the spinlock is blocked, and all CPUs are consumed spinning on that same lock.

It is reasonable to place this same restriction on the linked list in this example, so that, by convention, it is illegal to block while traversing the linked list, just as it would be if the list were guarded by a reader-writer spinlock. To avoid blocking in a preemptible environment, preemption must be disabled during the traversal. In both preemptible and non-preemptible environments, the traversal must not invoke any primitives that block, for example, primitives that wait for I/O completion. Note that any blocking operation results in a context switch.

Given this convention, Figure 2.24 illustrates how to determine when it is safe to return element B to the freelist.

1. Initially, the list contains elements A, B, and C in that order.

2. Element A's successor pointer is updated to point to Element C. At this point, a reader referencing element C may or may not have passed through element B, as indicated by the shaded arrows emanating from element A.

3. Once every CPU executes a context switch, there can no longer be any references to element B.

4. Element B may now be returned to the freelist.

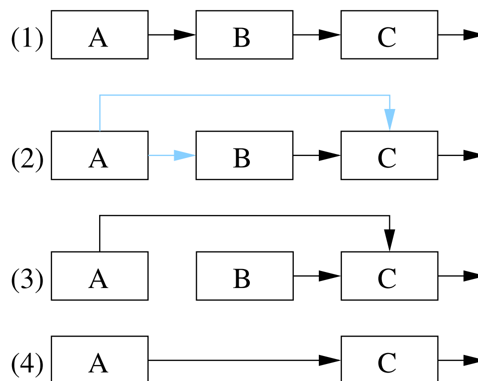


Figure 2.24: Atomic Deletion From a Linked List

The reasoning behind this procedure is as follows:

1. It is illegal to block while traversing the linked list.
2. It is illegal to hold a reference to an element of the linked list from one traversal to the next. Note that this convention is followed in traditional locking designs as well.
3. It is impossible to gain a new reference to element B once it has been unlinked from the list.
4. Because, by convention, preemption must be disabled during a traversal, it is not possible for a thread to migrate from one CPU to another without blocking.
5. Blocking on a given CPU forces that CPU to undergo a context switch.
6. Because it is illegal to block while traversing the linked list, if a CPU undergoes a context switch, it cannot be traversing the linked list.
7. Because it is illegal to hold a reference to an element of the linked list from one traversal to the next, if a CPU undergoes a context switch, it cannot be holding a read-side reference to any element of the list.[6]
8. Therefore, if a given CPU undergoes a context switch after element B has been unlinked from the list, that CPU cannot possibly be holding a read-side reference to element B.
9. Once all CPUs have undergone context switches, there can be no more read-side references to element B.

So, provided the requisite conventions are followed, once all CPUs have been observed to undergo at least one context switch, element B may safely be returned to the free pool. Checking for context switches will add overhead to updates, but if updates are rare, this overhead will be more than made up for by the fact that readers can omit expensive synchronization operations.

But can deferred destruction be exploited by real algorithms, to say nothing of real operating-system kernels?

[6]Note that the updater needs to retain a reference to the element in order to free it.

2.2.20 Deferred Destruction

Deferred destruction was first described in 1980, when Kung and Lehman [56] recommended use of a garbage collector to defer destruction of nodes in a parallel binary search tree in order to simplify its implementation. This good advice does not help much in environments that lack a garbage collector, such as most operating-system kernels. Even environments possessing garbage collectors typically impose significant overhead on readers, for example to maintain reference counts. In addition, it is not clear how great a performance increase can be obtained from a binary search tree given the frequent updates to the root node, in light of the high cost of pipeline stalls and memory latency.

In 1982, Manber and Ladner [61, 62] recommended deferring destruction until all threads running at that time have terminated. This recommendation is not helpful in operating-system-kernel or server environments, where threads are explicitly designed to *not* terminate. Nonetheless, this approach does eliminate all read-side synchronization overhead. However, it again is not clear how great a performance increase can be obtained from parallel search trees given the high cost of pipeline stalls and memory latency.

In 1986, Hennessy, Osisek, and Seigh [39] introduced passive serialization, which is a deferred-destruction mechanism that relies on the presence of "quiescent states" in the VM/XA hypervisor that are guaranteed not to be referencing the data structure. However, this mechanism was not optimized for the memory latency and pipeline-stall overheads found on modern computer systems, which is not surprising given that these overheads were not so expensive at that time. A similar mechanism is discussed in Appendix C.2.3 on Page 336, and it does in fact suffer from these overheads, which may explain why passive serialization was not applied very widely in VM/XA. Nonetheless, passive serialization appears to be the first deferred-destruction mechanism to be used in production.

In 1990, Pugh [93] noted that explicitly tracking which threads were reading a given data structure permitted deferred free to operate in the presence of non-terminating threads. However, this explicit tracking imposes significant read-side overhead, which is anything but desirable in read-mostly situations. Although this algorithm takes pains to avoid write-side contention and parallelize the other write-side overheads by providing

a fine-grained locking design, the high costs of pipeline stalls and memory latency make it unclear how much of the performance advantage reported in 1990 remains in 2004.

At about this same time, Adams [1] described "chaotic relaxation", where the normal barriers between successive iterations of convergent numerical algorithms are relaxed, so that iteration n might use data from iteration $n-1$ or even $n-2$. This introduces error, which typically slows convergence and thus increases the number of iterations required. However, this increase is sometimes more than made up for by a reduction in the number of expensive barrier operations, which are otherwise required to synchronize the threads at the end of each iteration. Unfortunately, chaotic relaxation requires highly structured data, such as the matrices used in scientific programs, and is thus inapplicable to most data structures in operating-system kernels.

In 1993, Jacobson [53] described what is perhaps the simplest deferred-free technique: simply waiting a fixed amount of time before freeing blocks awaiting deferred free.[7] This works well if there is a well-defined upper bound on the length of time that reading threads can hold references, as there might well be in hard real-time systems. However, if this time is exceeded, perhaps due to preemption, excessive interrupts, or larger-than-anticipated load, memory corruption can ensue, with no reasonable means of diagnosis. Jacobson's technique is therefore inappropriate for use in production operating-system kernels.[8]

In 1995, Pu et al. [91] applied a technique similar to that of Pugh's read-side-tracking to permit replugging of algorithms within a commercial Unix operating system. However, this replugging permitted only a single reader at a time. The following year, this same group of researchers extended their technique to allow for multiple readers [23]. Their approach requires memory barriers (and thus pipeline stalls), but reduces memory latency, contention, and locking overheads.

Finally, in 2002, Michael [86, 87] presented techniques that defer the destruction of data structures to simplify NBS synchronization. In particular, this technique eliminates locking, reduces contention, reduces memory latency for readers, and parallelizes pipeline stalls

[7] Jacobson did not describe any write-side changes he might have made in this work using SGI's Irix kernel. Aju John published a similar technique in 1995 [54].

[8] Except for kernels that can provide hard real-time response guarantees for all operations.

and memory latency for writers. However, these techniques still impose significant read-side overhead, particularly in the form of memory barriers, as discussed in Section 2.2.14 on Page 51. Researchers at Sun worked along similar lines concurrently [43, 45].

These mentions of deferred destruction, though important, did not present an efficient mechanism for determining how long to defer destruction in SMMP environments lacking a garbage collector but with long-running threads. The lack of such a mechanism prevented the benefits of deferred destruction from being realized.

This gap was filled by this author's work on DYNIX/ptx's RCU [81, 108] and by the Tornado and K42 research operating systems' "generations", each of which provides a NUMA-optimized specialized garbage collector to implement deferred destruction in an easy-to-use fashion. In both cases, the key idea is a straightforward API that allows data to be queued for later destruction, so that the programmer need not be concerned with the details of the actual destruction deferral implementation. The properties of the write-side access are dictated by the write-side synchronization mechanism, but typically parallelize pipeline stalls, memory latency, and locking while reducing contention. In all three operating systems, deferred destruction enables synchronization-free read-only access. However, this use was restricted to a few subsystems in the case of DYNIX/ptx and to existence locking in the case of K42 [30]. Widespread use was hindered by the lack of a robust set of design patterns for the typical practitioner to follow.

2.3 Discussion

This chapter has reviewed a number of synchronization mechanisms, each of which has its own strengths and weaknesses. This section reviews these strengths and weaknesses with an eye to creating a synchronization mechanism that is well-suited to the many read-mostly data structures found in operating-system kernels such as Linux. Section 2.3.1 discusses relevant hardware costs and trends, and their relationship to synchronization-mechanism design. Section 2.3.2 discusses the principles learned from this chapter's review of existing synchronization mechanisms. Finally, Section 2.3.3 summarizes the synchronization-mechanism attributes desired for guarding read-mostly data structures in operating-system

kernels.

2.3.1 Costs

The preceding sections have shown that effective use of synchronization mechanisms can be quite complex. Such use must be cognizent of the four fundamental costs incurred by SMMP software:

1. Instruction execution
2. Pipeline stalls
3. Memory latency
4. Contention

Of these, the relative costs of pipeline stalls and especially of memory latency have been steadily increasing. In contrast, Moore's Law has been sharply and steadily decreasing instruction-execution costs. Contention is a function of the design, with the goal being unabashed embarrassing parallelism.

2.3.2 Principles

The review of related work has brought out the following observations:

1. Confining control of interrupt and preemption disabling to the operating-system kernel helps prevent denial-of-service attacks.
2. Layered design and implementation simplifies synchronization primitives.
3. Reader-writer semantics, resulting in read-side and write-side critical sections, permit read-side critical sections to proceed in parallel.
4. Incurring memory latency and pipeline stalls in parallel rather than serially can improve both performance and scalability.
5. Asymmetric locking primitives, which favor the performance of the common case at the expense of increased performance for less-common cases, can greatly improve performance.

6. Data placement into cachelines can affect performance.

7. Eliminating locking eliminates lock contention.

8. Eliminating cache-line transfers reduces memory-latency overhead.

9. Even in the absence of locks, repeated atomic-operation retries can degrade performance.

10. Synchronization primitives that permit incremental adoption are more easily incorporated into large production projects.

11. Exploiting operation semantics can increase scalability and performance.

12. Maintaining multiple versions of portions of a data structure can simplify the interaction between updates and read-only accesses, permitting greater concurrency as well as lock-free read-side access.

13. Deferring destructive operations can greatly simplify read-only accesses to data structures and enhance read-side performance.

14. Hiding complex operations behind atomic "commit points" can reduce the need for readers to synchronize with updaters.

15. Use of specialized garbage collectors eases the use of deferred destruction.

In addition, this review of related work identified the following design patterns, which are discussed in more depth in Section 5.2 on Page 138:

1. Code Locking.

2. Reader/Writer Locking.

3. Critical-Section Partitioning.

4. Data Locking.

These observations and patterns can be abstracted into three basic (and unsurprising) principles:

1. Avoid expensive operations, namely those incurring pipeline-stall and memory-latency overheads.

2. Architect, design, and implement algorithms that permit fully parallel useful execution, thereby avoiding contention.

3. Architect, design, and implement algorithms that meet software-engineering needs, including simplicity, resilience against denial-of-service attacks, and tolerance for incremental adoption.

2.3.3 Attributes

The attributes desired of a synchronization mechanism depend on the usage and workload. In this dissertation, the focus is on read-mostly data structures in operating-system kernels. These kernels run on SMMP computer systems, contain an abundance of read-mostly data structures, and permit partial ordering of many operations. Taking these considerations into account yields the following desirable attributes for synchronization mechanisms:

1. Partial ordering, allowing additional parallelism. In contrast, total ordering inherently limits parallelism.

   (a) Concurrent reads, allowing partial ordering specifically among read-only accesses.

   (b) Reads concurrent with writes, allowing partial ordering specifically among reads and writes. Note that allowing reads and writes to proceed concurrently means that readers can see stale data. Although many algorithms tolerate stale data, others do not. However, Section 5.3 on Page 159 describes some methods of transforming algorithms into a form that tolerates stale data.

   (c) Concurrent writes, allowing partial ordering specifically among updates.

2. Asymmetric distribution of synchronization overhead, in this case, favoring readers over writers.

3. Reads and writes either avoiding or parallelizing:

(a) pipeline stalls

(b) memory latency

(c) contention

(d) locking

Note that the memory latency, pipeline stalls, and contention called out in item 3 are exactly the high-cost overheads noted in Section 2.2.3. Locking is called out separately because it is a common operation that incurs all three of these types of overhead.

A summary of the attributes of the synchronization mechanisms presented in this chapter is shown in Table 2.2. In this table, an empty cell indicates that the corresponding synchronization mechanism either has no effect or degrades the corresponding attribute. Different letters indicate different types of support, as follows:

A Avoids the specified type of overhead.

P Parallelizes the specified type of overhead.

Y Supports the specified attribute.

A lower-case letter indicates partial support. For example, data locking provides concurrent reads to different data items, but only if they hash to different buckets in the hash table. In contrast, brlock provides concurrent reads regardless of which elements are being accessed. Therefore, brlock gets an uppercase "Y" for partial ordering, while data locking gets a lowercase "y".

Code locking, reader-writer locking, data locking, data and reader-writer locking, and brlock all have the advantage of familiarity, and, in particular, that writers block readers. This last feature helps neither performance nor parallelism, as we have seen, but can reduce code complexity in some cases. The non-blocking-synchronization row reflects the attributes of generalized NBS, since although simple NBS can be quite efficient, the wide variety and extreme specialization of simple NBS techniques defy reasonable classification. The question marks in the "Exploiting Semantics" row indicate that the properties depend on the exact semantics being exploited, while those in the "Deferred Destruction" row indicate that these attributes depend on the chosen write-side synchronization mechanism.

Table 2.2: Attributes of Synchronization Mechanism

					Reads:				Writes:			
	Concurrent Reads	Reads Concurrent With Writes	Concurrent Writes	Asymmetric	Pipeline Flushes	Memory Latency	Contention	Locking	Pipeline Flushes	Memory Latency	Contention	Locking
Code Locking												
Reader-Writer Locking	Y			y			A					
Data Locking	y	y	y		p	p	a	p	p	p	a	p
Data & RW Locking	Y	y	y	y	p	p	A	p	p	p	a	p
brlock	Y			Y	P	A	A	P				
OS Partitioning	y	y	y		p	p	a	p	p	p	a	p
Non-Blocking Synchronization	y	y	y		p	p	a	p	p	p	a	p
Exploiting Semantics	y	y	y	Y	?	?	?	?	?	?	?	?
Deferred Destruction	Y	Y	?	Y	A	A	A	A	?	?	a	?

The deferred-destruction class of synchronization mechanisms has many attractive properties. The next chapter explores these properties in more depth, in the course of presenting an overview of RCU, which is a promising deferred-destruction-based synchronization mechanism that has recently been accepted into the Linux 2.6 kernel.

Chapter 3

RCU Overview

This chapter provides an overview of RCU, which is a promising deferred-destruction-based synchronization mechanism that was implemented in the DYNIX/ptx operating-system kernel by the author in collaboration with Jack Slingwine [81]. As part of DYNIX/ptx, RCU has seen use in mission-critical datacenter environments running large database servers.[1] The author later acted as architect for the implementation of RCU in Linux, which led to RCU being accepted into the Linux 2.5.43 kernel [121]. Variants of RCU have been independently implemented in the K42 and Tornado research operating systems and in IBM's mainframe VM/XA virtual-machine monitor.

This chapter presents an introduction to RCU in Section 3.1, describes how RCU solves concurrency problems in Section 3.2, presents a conceptual overview of RCU in Section 3.3, demonstrates example uses of RCU in Section 3.4 and summarizes analogies and design patterns in Section 3.5. This chapter is adapted and expanded from material that this author previously published [11, 71, 73, 78].

3.1 Introduction to RCU

RCU is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur zero synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance

[1] In 1999, seven of the ten largest Oracle installations used RCU as implemented in the DYNIX/ptx kernel.

for read-only workloads on most architectures. Write-side critical sections must therefore incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to provide for orderly updates.

Readers must provide a signal enabling writers to determine when it is safe to complete destructive operations, but this signal may be deferred, permitting a single signal operation to serve multiple read-side RCU critical sections. RCU typically signals writers by non-atomically incrementing a local counter, which is an extremely inexpensive operation.

These read-side signals are observed by a specialized garbage collector, which carries out destructive operations once all readers have signalled that it is safe to do so. Garbage collectors are typically implemented in a manner similar to a barrier computation, or, on NUMA systems, a combining tree. Production-quality garbage collectors batch destructive operations, so as to amortize their overhead over many write-side update operations.

RCU has the attributes listed in the "Deferred Destruction" row of Table 2.2 on Page 69. These attributes are further expanded on in Table 3.2 on Page 97.

RCU provides concurrent reads, in fact, since readers do not use any synchronization mechanism, there is no way for readers to avoid concurrency. For the same reason, RCU provides concurrent reads and writes. RCU does not specify whether writers may run concurrently with each other; write-side concurrency depends instead on the chosen write-side synchronization mechanism. As noted earlier, RCU is maximally asymmetric, favoring readers to the greatest extent possible. The fact that RCU read-side critical sections use no synchronization mechanisms means that there is no overhead due to pipeline stalls, memory latency, contention, or locking for readers. Write-side overhead depends on the chosen write-side synchronization mechanism, but contention is reduced due to the fact that readers do not use any synchronization mechanisms.

3.2 How RCU Solves Concurrency Problems

This section describes the concurrency problems that RCU must solve, and describes the solutions in detail for the concrete case of non-preemptive operating-system kernels. This description sets the stage for a more abstract view of RCU given in later sections. This abstract view provides the conceptual tools needed to apply RCU to other environments, such as preemptive operating-system kernels and user-mode applications.

RCU is a reader-writer synchronization mechanism:

- Readers need not execute any explicit synchronization instructions, but must also avoid executing any context switches while traversing any RCU-protected data structures. However, readers must eventually execute a context switch after completing a given traversal.

- Writers must maintain multiple versions of data that is being updated so that readers can avoid encountering fatal inconsistencies, such as pointers to free memory.

RCU implementations must solve the following problems:

1. providing for readers and writers,

2. handling multiple versions; reclaiming old versions when it is safe to do so,

3. writer-writer synchronization,

4. reader-writer synchronization, and

5. interaction with weak memory-consistency semantics.

Each of these topics is addressed in the following sections.

3.2.1 Readers and Writers

A read-mostly data structure will benefit from a synchronization mechanism that classifies each access as either a “reader”, which does not perform significant changes to state, or

a "writer", which performs significant updates. Note that readers might update statistical counters or modify per-CPU private state, but would not normally make significant changes visible to other participants.

The key distinction between readers and writers is that readers are permitted to execute concurrently, while writers are not. Permitting readers to execute concurrently provides good scalability for read-mostly data structures.

The key difference between RCU and reader-writer locking is that RCU's read-side critical sections need not execute any synchronization instructions of any kind: no locks, no atomic instructions, no writes to shared memory, and, on almost all CPUs, no pipeline stalls. Read-side RCU critical sections therefore do not incur any pipeline-stall, memory-latency, or contention overhead. As such, RCU takes reader-writer asymmetry to its logical extreme, exacting no overhead from readers, but significant overhead from writers.

RCU is also the next step in the progression from exclusive locking to reader-writer locking to RCU, where exclusive locking enforces the strongest causal ordering and RCU the weakest. In exclusive locking, the execution sequence of all critical sections corresponding to a given lock are totally ordered, as illustrated by Figure 3.1. All operations within a given critical section execute after completion of the predecessor critical section and before the beginning of the successor critical section.[2]

In reader-writer locking, the execution sequence of write-side critical sections corresponding to a given lock are totally ordered, but the execution sequence of read-side critical sections is unordered, in fact, as noted earlier, read-side critical sections may run concurrently. However, read-side critical sections may *not* run concurrently with write-side critical sections. Therefore, it is possible to group the read-side critical sections into sets, with each set comprising all the read-side critical sections that execute between a pair of consecutive write-side critical sections. Then each write-side critical section is followed by either another write-side critical section or by a set of read-side critical sections. All sets of read-side critical sections are by definition always preceded and followed by write-side

[2]The ordering of the operations *within* a given critical section depends on the system's memory-ordering model, and ranges from totally ordered in sequentially consistent systems to unordered in release-consistent systems.

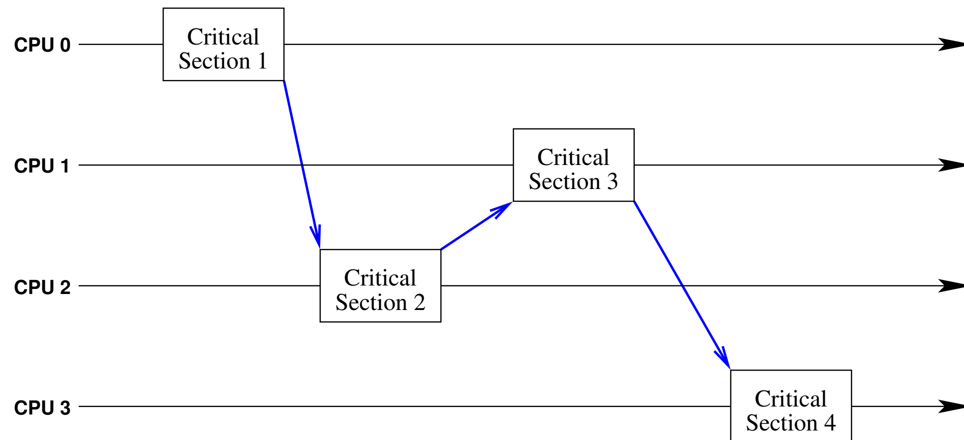


Figure 3.1: Causal Ordering for Exclusive Locking

critical sections. The write-side critical sections and the sets of read-side critical sections are totally ordered, as illustrated in Figure 3.2.

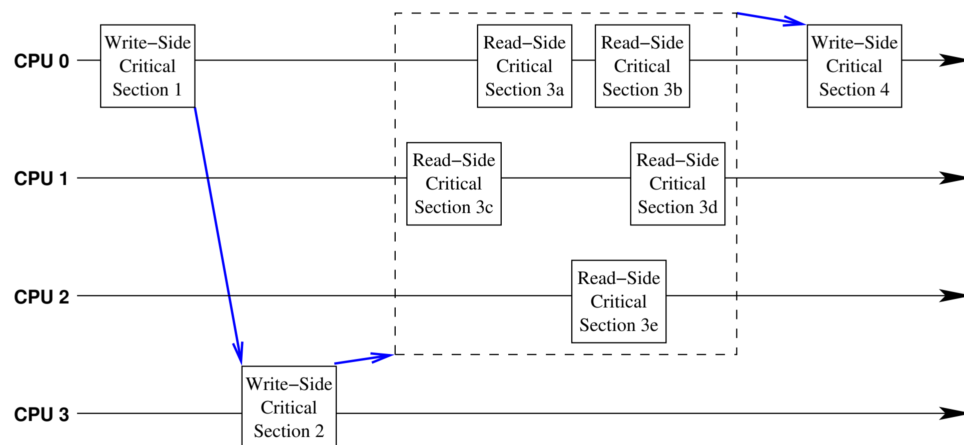


Figure 3.2: Causal Ordering for Reader-Writer Locking

In contrast, RCU read-side critical sections are unordered with respect both to write-side critical sections and with respect to each other, as illustrated in Figure 3.3. The write-side critical sections are still totally ordered, under the assumption that they are guarded by a single exclusive lock. The write-side critical sections could instead make use of non-blocking synchronization or reader-writer locking.

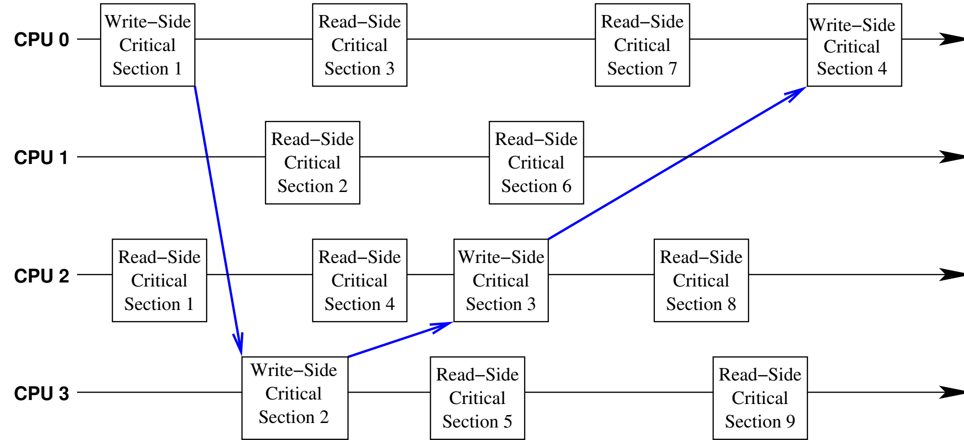


Figure 3.3: Causal Ordering for RCU

3.2.2 Multiple Versions

Since RCU readers are not executing any sort of synchronizing instructions within their critical sections, writers must typically maintain multiple versions of the data structures to avoid inducing read-side failures. A given version may be removed upon completion of all read-side RCU critical sections that were in progress at the creation of the replacement version.

Figure 3.4 shows an example of how these multiple versions might be handled, illustrated by a `head` pointer to a dynamically allocated data item. Step 1 shows the initial list, with the `head` pointer referencing version 1 of the dynamically allocated structure. Step 2 shows the list after the first update, which creates version 2 of the structure. The light arrows indicate that there might still be readers referencing version 1 of the structure. Step 3 shows the list after the second update, which creates version 3 of the structure, again, with the light arrows indicating that there might still be readers referencing the first two versions of the structure. Step 4 shows the list after all readers referencing version 1 of the structure have exited their read-side RCU critical sections. A mechanism for determining that all readers referencing a given structure have exited their read-side RCU critical sections has been introduced in Section 2.2.19 on Page 59, and similar mechanisms will be discussed in more detail in Section 3.2.4 later in this chapter and in Chapter 4.

Step 5 shows the list after version 1 of the structure has been freed. Step 6 shows the list after the third update, which creates version 4 of the structure, with the light arrows again indicating that there might still be readers referencing version 2 of the structure. This step illustrates an important point–readers will not necessarily exit their read-side RCU critical sections in order. Step 7 shows the list after all readers referencing versions 2 and 3 of the structure have exited their read-side RCU critical sections. Step 8 shows the list after versions 2 and 3 of the structure have been freed.

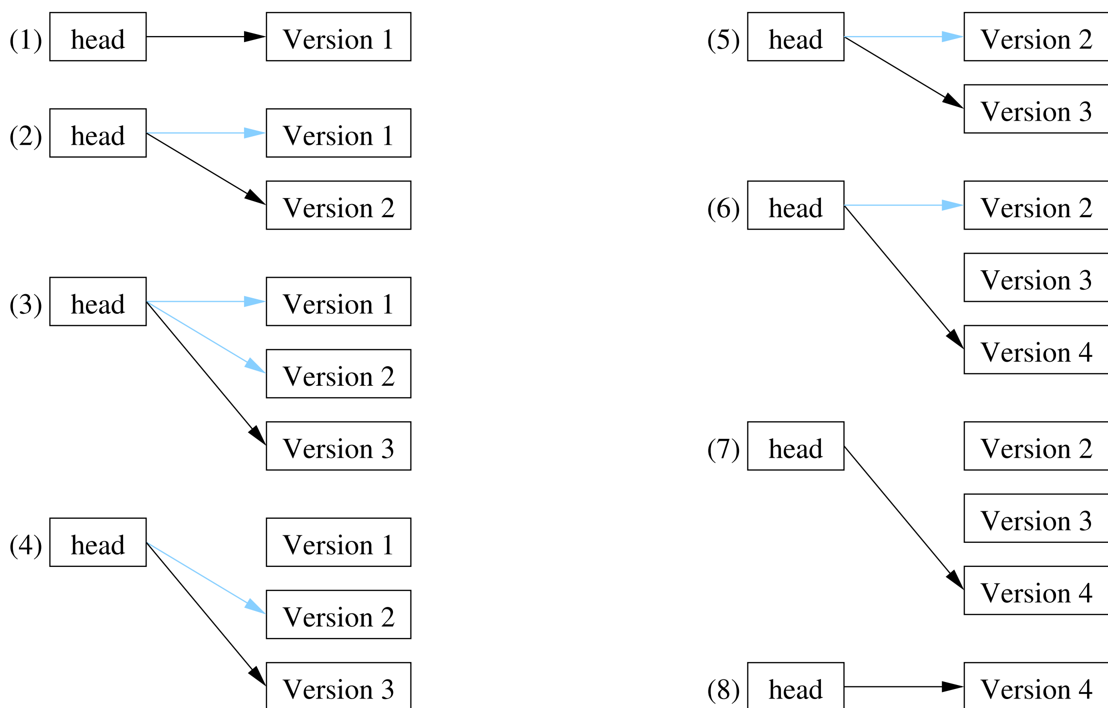


Figure 3.4: RCU Multiple Version Handling

Figure 3.5 shows this same sequence of events on a timeline with time proceeding downwards as indicated by the arrow. The dashed vertical arrows show how long an obsolete version of the data structure is retained, and the length of time that readers are accessing a given obsolete version of the structure is indicated by the arrows around the "readers" annotation. Note that it is quite possible that a given version of the data structure might not be referenced by a reader at the time of its replacement, as indicated

by version 3 of the data structure lacking a "readers" indicator. At the end of the timeline, only version 4 of the data structure remains.

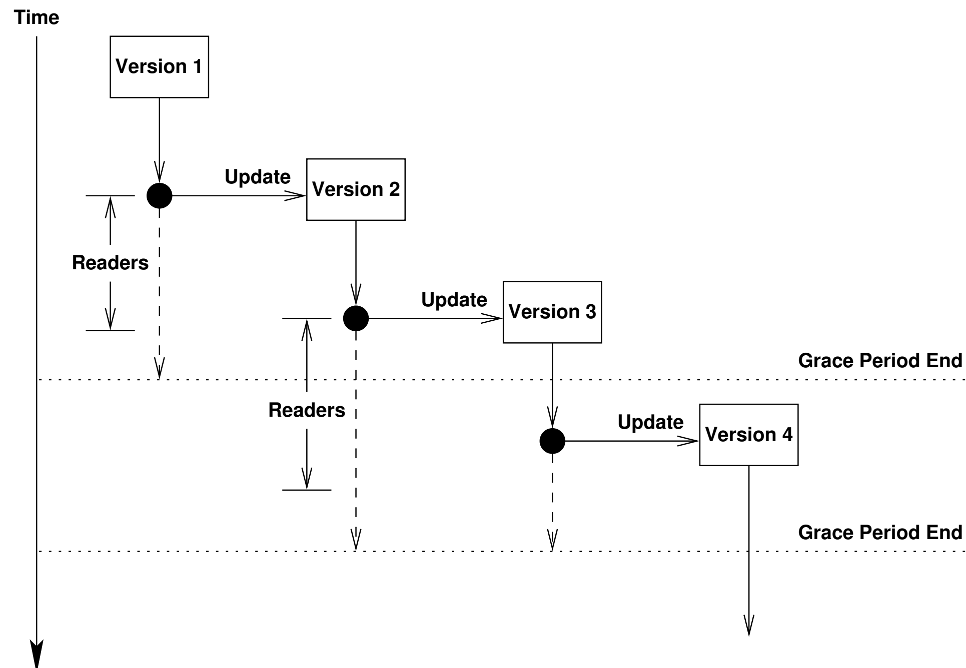


Figure 3.5: RCU Multiple Version Timeline

This maintenance of multiple versions is the fundamental mechanism that allows RCU readers to preceeds without synchronization, and each such version corresponds to an update. Each update is split into two distinct operations: (1) creation of a new version of an element, and (2) deletion of the previous version of that element. The first operation can be done atomically, without involving the readers, as will be shown in Section 5.3.3. The second operation can be deferred indefinitely without affecting the correctness of the readers. However, writers must synchronize with each other, as described in the next section.

3.2.3 Writer-Writer Synchronization

RCU does not specify what type of writer-writer synchronization should be used. In most cases, locking is used to synchronize writers, most frequently code locking or data locking.

However, any other synchronization mechanism may be used, for example, Section 6.7 on Page 221 demonstrates use of non-blocking synchronization for writer-writer synchronization in conjunction with RCU for reader-writer synchronization.

3.2.4 Reader-Writer Synchronization

Given that RCU does not require readers to execute any synchronization instructions, RCU reader-writer synchronization may seem at first glance to be paradoxical. However, synchronization *is* required in order to prevent writers from prematurely destroying data elements that readers are still referencing. This apparent paradox is resolved by the fact that the synchronization between RCU readers and writers is indirect, as described below, as well as in the example in Section 2.2.19 on Page 59. In addition, read-side signalling may be batched, so that one such signal can serve for an arbitrary number of preceding read-side RCU critical sections.

In the Linux kernel, RCU imposes the same rules on RCU read-side critical sections that a reader-writer lock imposes on its read-side critical sections:

1. preemption is disallowed,
2. blocking is disallowed,
3. references to protected data structures cannot be held outside of a critical section, and
4. execution must complete in a reasonable length of time, where "reasonable" is defined with respect to the workload and environment. For example, a hard realtime system would have a much more restrictive definition of "reasonable" than would a web-serving system that is used across a long-latency satellite link.

The first two rules mean that context switches cannot occur in an RCU read-side critical section, the third means that all references to the protected data structure must be dropped upon exit from an RCU read-side critical section, and the last rule means that RCU read-side critical sections must complete reasonably quickly.

Despite the fact that read-side RCU and reader-writer-lock critical sections follow the same rules, RCU differs in that while reader-writer lock must directly signal writers at the end of each and every read-side critical section, RCU may defer such signals, permitting a single signal to notify writers of the completion of an arbitrarily large number of read-side RCU critical sections. RCU's batching of read-side notifications greatly reduces the synchronization overhead incurred by readers.

This deferred synchronization between RCU readers and writers then proceeds as follows:

1. Since context switches cannot occur in RCU read-side critical sections in the non-preemptive case, any thread[3] that has blocked must have completed all preceding RCU read-side critical sections.

2. Since RCU read-side critical sections must drop all references to RCU-protected data structures upon exit, any thread not running in an RCU read-side critical section cannot hold any references to any RCU-protected data structures.

3. Since blocked threads cannot be running in an RCU critical section, and since threads not running in an RCU critical section cannot hold references to RCU-protected data structures, blocked threads cannot hold references to RCU-protected data structures.

4. If an element has been removed from an RCU-protected data structure, then there is no longer a path to that element, and threads cannot subsequently obtain any new reference to it. Once all threads referencing it from a read-side RCU critical section drop their references, the element will be "dead" in that its value can no longer influence subsequent read-side RCU critical sections.

5. Since blocked threads cannot hold references to RCU-protected data structures, and since threads cannot obtain any new references to any element that has previously been removed from an RCU-protected data structure, a thread that is blocked after an element has been removed from an RCU-protected data structure cannot

[3]Here, "thread" is used in a general sense, including things like processes, tasks, interrupt handlers, event handlers, coroutines, and transactions.

subsequently obtain a reference to that element.

6. Once all threads have been observed in the blocked state after a given element has been removed from an RCU-protected data structure, no thread in an RCU read-side critical section can be holding a reference to that element.[4]

One issue with this approach is that a given system might be running literally millions of threads. Explicitly checking the state of each thread will incur far too much overhead, overwhelming the performance gains due to eliminating all synchronization instructions from the read-side RCU critical sections. In a non-preemptive system, one way to address this issue is to note that only those threads currently running can possibly be in a read-side RCU critical section–all others must by definition be blocked. Therefore, it is sufficient to check only the threads currently running on a CPU, since any thread not currently running on a CPU must be blocked. Since most systems have many more threads than CPUs, this greatly reduces the overhead of checking for the completion of all read-side RCU critical sections.[5]

This mapping, coupled with the deferred synchronization means that an element in an RCU-protected data structure may be safely deleted using the following procedure:

1. Remove an element from the RCU-protected data structure, leaving intact any fields referenced by RCU read-side critical sections.

2. Wait until all CPUs have each executed at least one context switch.

3. Perform any required destructive operations, such as returning the element to the freelist.

This indirect synchronization between RCU readers and writers ensures that readers will no longer be referencing any element by the time that it is destroyed in step 3 of the above deletion procedure.

Operating systems that maintain per-CPU context-switch counters can use a particularly straightforward implementation of the second step, as follows:

[4]Of course, the thread removing the element presumably retains a reference in order to free it at the end of a grace period.

[5]A similar technique may be applied in preemptive systems, as is discussed in Appendix C.3 on Page 343.

1. Take a snapshot of each CPU's context-switch counter, placing the result in a local array of counters.

2. While at least one of the per-CPU context-switch counters is equal to its counterpart in the local array, block for a short time interval.

Eventually, each CPU will undergo a context switch and increment its per-CPU context-switch counter. Once that happens, each CPU's context-switch counter will differ from its counterpart in the local array, and the above procedure will terminate. Although this procedure is quite straightforward, it is also rather inefficient. A number of higher-performance implementations are described in Chapter 4.

Note that this indirect synchronization mechanism may be used for purposes other than deletion. For example, consider a log buffer whose individual elements are allocated and filled in by RCU read-side critical sections. Once all elements have been allocated and a grace period has elapsed, all of the log buffer's elements are guaranteed to be filled in, so it is safe to write the entire log buffer to stable storage.

3.2.5 Weak Memory-Consistency Semantics

Modern CPUs have weak memory-consistency semantics, so that the order of reads and writes may be changed by the underlying hardware, and so that different CPUs may disagree on the order of read and write operations [33]. These weak semantics are motivated by hardware performance considerations and by the observation that the individual CPUs are normally operating either on private data or on data for which they hold a lock, so that the order in which a given CPU's memory operations occur as observed by other CPUs is normally irrelevant. CPUs provide *memory barrier* instructions to permit software to enforce ordering where needed, for example, when acquiring or releasing a lock so that the critical section does not "bleed out" into the surrounding code as described in Section 2.2.10.

RCU writers must execute memory barriers as needed to ensure that RCU readers always see a consistent view of RCU-protected data structures. For example, an RCU writer must use a procedure similar to the following when adding a new element to an

RCU-protected data structure:

1. Allocate and initialize the new element, including pointers to its neighbors-to-be.

2. Execute a memory-barrier instruction.

3. Update pointers in existing elements in the RCU-protected data structure to point to the new element.

In this procedure, the memory barrier ensures that other CPUs see all of the memory writes that initialize the new element in step 1 before any of the pointer updates in step 3 of this procedure.

The DEC/Compaq/HP Alpha has especially weak memory-barrier instructions [19]. The Alpha therefore requires special handling, which is discussed at length in Appendix B on Page 322.

3.3 Conceptual Overview of RCU

Section 3.3.1 presents an RCU glossary and Section 3.3.2 gives an overview of the tradeoffs encountered in designing RCU infrastructure.

3.3.1 RCU Glossary

The following definitions provide a vocabulary for the concepts underlying RCU. Keeping these definitions clearly in mind allows one to more easily apply RCU in different environments. Figure 3.6 serves to illustrate these definitions, and is discussed at the end of this section.

Thread: A thread is a locus of execution. "Thread" is used in a general sense that includes things like processes, tasks, interrupt handlers, event handlers, coroutines, and transactions.

Live Variable: A variable that might be accessed before it is next modified, so that its current value has some possibility of influencing future execution state.

Dead Variable: A variable that will be modified before it is next accessed, so that its current value cannot possibly have any influence over future execution state.

Critical Section: A region of code whose accesses to shared memory are protected from outside interference through use of some synchronization mechanism, such as spinlocks or RCU. For example, line 6 of the `search()` function in the upper-left-hand cell of Table 3.1 on Page 92 is a critical section protected by the `listmutex` reader-writer lock.

Read-Side Critical Section: A region of code whose accesses to shared memory are protected from outside interference, again through use of some synchronization mechanism, but which permits multiple concurrent readers. Line 6 of the `search()` function noted above is a read-side critical section; see the `delete()` functions in that same table for an example of a non-read-side critical section. In the case of RCU, data referenced by RCU read-side critical sections is protected from being freed.

Temporary Variable: A variable that is only live inside a critical section. One example of a temporary variable is an auto variable used as a pointer while traversing a linked list, such as the variable `p` in the `_search()` function in Figure 3.8 on Page 91.

Persistent Variable: A variable that is live outside of critical sections. One example would be the header for a linked list, such as the variable `head` in the `_search()` function noted above. Although it is possible for the same variable to be temporary sometimes and persistent at other times, this practice can lead to confusion, so is not generally recommended. Relying on the register-allocation capabilities of modern optimizing compilers is usually a far better strategy.

Quiescent State: A thread-execution state during which no references to any RCU-protected data structures are held. There are two types of quiescent state, a "candidate quiescent state" and an "observed quiescent state". This dissertation typically uses "quiescent state" in the "observed quiescent state" sense. It is used in the "candidate quiescent state" sense when designing read-side RCU algorithms.

Candidate Quiescent State: A point in the code where all of this thread's temporary variables that were previously in use in a critical section are dead. Any point that lies outside of all RCU read-side critical sections is a candidate quiescent state.

It is important to note that candidate quiescent states are quiescent with respect to a selected set of data structures. It is possible to have multiple sets of candidate quiescent states with respect to multiple sets of data structures in the same operating-system kernel. For example, recent patches for the Linux 2.6 kernel have created a separate set of quiescent states suitable for use by some critical portions of TCP/IP, this new set being better able to withstand some types of denial-of-service attacks.

Observed Quiescent State: Taking each and every candidate quiescent state into account would impose severe overhead or would require special hardware support. However, some candidate quiescent states may be observed particularly inexpensively, since they are associated with a persistent operating-system state change. These are known as "observed quiescent states". For example, in a non-preemptive Linux kernel, context switch is an observed quiescent state since a counter is incremented on each context switch. In a preemptive Linux kernel, `rcu_read_lock()` and `rcu_read_unlock()` suppress preemption for short read-side critical sections, so that context switch is still an observed quiescent state with respect to these read-side critical sections. In either case, a read-side RCU critical section is not permitted to contain any operation that serves as an observed quiescent state.

Although there are implementations of RCU that do not require preemption to be suppressed, for example, by considering only *voluntary* context switches to be observed quiescent states [30, 78], they can be prone to excessively long grace periods. Nevertheless, the K42 and Tornado research operating systems have obtained good results with this approach [9].

As noted earlier, most of this dissertation uses "quiescent state" in the "observed quiescent state" sense.

Grace Period: A time interval during which all threads pass through at least one quiescent state. The key property of a grace period is that any temporary variables used by any RCU read-side critical section at the beginning of that grace period are dead at least once during that grace period. These variables therefore cannot possibly have any direct effect after the end of the grace period, and, in particular, they cannot possibly be referencing any element that was removed from its data structure at any time prior to the start of the grace period. Note that any time interval containing a grace period is itself a grace period.

Figure 3.6 illustrates these terms. Threads A-H execute on CPUs 0-3, with time progressing from left to right. The boxes denote a read-side RCU critical section, so that the box labelled "A1" on CPU 0 is thread A's first read-side RCU critical section, "A2" also on CPU 0 is thread A's second, "A3" on CPU 1 is thread A's third, and so on. The line segments between the boxes represent continued thread execution. Since this execution is outside of any read-side RCU critical section, each point on these lines represents a candidate quiescent state. This means that all temporary variables used in any of the read-side RCU critical sections must be dead whenever execution is proceeding along one of these line segments. In contrast, persistent variables may remain live throughout. The black circles denote context switches, and these context switches are the observed quiescent states.

Suppose that CPU 3 is updating an RCU-protected data structure at the time indicated by the leftmost dotted line. All read-side RCU critical sections in progress at that time have completed by the time indicated by the middle dotted line, so that the time between these two lines is the minimum grace period, denoted in the figure by "Min GP". However, only the context switches are observed, so the grace period is not marked complete until all CPUs pass through a context switch. The last CPU to do so is CPU 2, as noted by the rightmost dotted line. Although the grace-period latency could be greatly reduced by observing all possible quiescent states, doing so would greatly increase grace-period detection overhead. For example, observing the end of each read-side RCU critical section would incur roughly as much overhead as does a reader-writer lock. This tradeoff between

overall efficiency and grace-period latency will be revisited in Section 8.3.3 on Page 279.

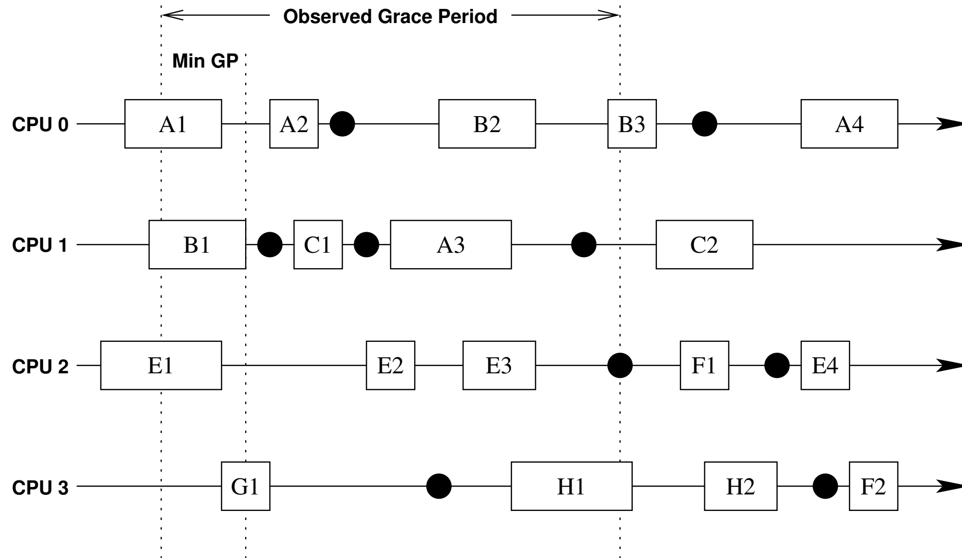


Figure 3.6: RCU Concepts

3.3.2 RCU Design Tradeoffs

The vocabulary defined in the previous section permits description of RCU independently of a given environment. This in turn permits architecting the RCU infrastructure to a particular environment and making the necessary design tradeoffs.

The first step is to identify an appropriate set of observed quiescent states. These observed quiescent states must be used infrequently enough that they can be forbidden from read-side RCU critical sections, which means that the programmer must have control of them. For example, it does not make sense to use context switch as a quiescent state in a preemptive operating system kernel unless there is some way to suppress preemption. However, it *does* make sense to use *voluntary* context switch as a quiescent state, as in the K42 research operating system (see for example Section 4.4.3 on Page 130). However, the chosen observed quiescent states should also be states that must be tracked anyway, for example, for system-monitoring or debugging purposes. Context switch meets both needs admirably in most non-preemptive operating systems. Example observed quiescent states

for selected applications include transaction boundary for a real-time database, request boundary for a web server, event termination for an event-driven system, and so on. If there are no such "natural" quiescent states, it is possible to create "artificial" quiescents states. However, this approach has the drawback of adding quiescent-state overhead that would not otherwise be present.

The key to a useful RCU implementation is a CPU-efficient mechanism for determining the required duration of the grace period, given the set of observed quiescent states. This mechanism is permitted to overestimate the grace-period duration, but the greater the overestimation, the greater the amount of memory that will be consumed by waiting deferred deletions, henceforth called "callbacks". There are a number of simple and efficient algorithms to determine grace-period duration, a number of which are reviewed in Chapter 4.

There are a number of design parameters for an RCU implementation:

1. Batching. Many implementations batch requests, so that a single grace-period identification can satisfy multiple requests. Batching is particularly important for implementations with heavyweight grace-period identification mechanisms. Batching can also increase performance due to increased cache locality of the RCU callback invocations compared to isolated execution of the corresponding code in non-RCU implementations.

2. Determining the length of the grace period. The simplest mechanisms force a grace period by a reschedule on all CPUs in non-preemptive kernels, as described in Section 4.3 on Page 108. However, this approach is relatively expensive, particularly if extended to cope with preemptible kernels. More efficient implementations use something like per-CPU quiescent-state counters to determine when the natural course of events has resulted in the expiration of a grace period.

3. Polling mechanism. Implementations that determine when a grace period has ended must use some mechanism to be informed of this event. Examples include:

   (a) Adding explicit checks to quiescent-state code, for example, *rcu-sched*'s hooks

in the Linux scheduler shown in Figure 4.26 on Page 127. Explicit checks allow fast response to quiescent states, but add overhead when no one is waiting for a grace period to complete.

(b) Adding counters to code corresponding to quiescent states, and using kernel daemons to check the counters periodically. This approach adds some complexity, but again greatly reduces the overhead when no one is waiting for a grace period to complete.

(c) As above, but use tasklets[6] instead of kernel daemons to do the checking. This further reduces the overhead, but uses more exotic features of Linux.

(d) As (3b) above, but use a per-CPU timer handler instead of tasklets to do the checking.

If the implementation directly executes a set of quiescent states, it must similarly use a mechanism for doing so:

(a) Scheduling a thread on each CPU in turn. This has the advantage of immediacy, but gains no performance benefit from batching, and cannot be used in any context where blocking is prohibited, such as the "bottom half" (BH) of a Linux device driver or from an interrupt-request (IRQ) handler.

(b) Reserving a kernel daemon that, upon request, schedules itself on each CPU in turn. This permits batching and use from BH and IRQ, but is more complex.

In either case, the end of the grace period is detected implicitly by the fact that the required set of quiescent states have been executed.

4. Request queuing. Requests may be queued globally or on a per-CPU basis. Grace periods must, of course, always be detected globally, but per-CPU queuing can reduce the CPU overhead incurred when enqueuing and dequeuing grace-period requests. This is a classic performance/complexity tradeoff. The correct choice depends on the workload.

[6] A "tasklet" is a Linux kernel mechanism that permits functions to be queued for later invocation in a controlled environment. This can be used, among other things, to avoid deadlock situations by deferring invocation of a given function to a later time when a conflicting lock as been released.

5. Quiescent state definition. For non-preemptive kernels, context switch is a popular choice. For preemptive Linux kernels (such as Linux 2.6), voluntary context switch may instead be used. Other environments might chose a different set of quiescent states.

6. Environments. If enqueuing of grace-period detection requests is prohibited in the BH or IRQ contexts, then more kernel functionality may be used to implement the enqueuing operation, and less overhead is incurred.

3.4 Examples

The following sections present simple examples of how RCU may be used. Section 3.4.1 describes how RCU may be used as a replacement for reader-writer locking, and Section 3.4.2 describes RCU's use of deferred destruction.

3.4.1 RCU Applied to Reader-Writer Locking

Splitting updates into two phases, so that destructive operations are deferred until a grace period has elapsed, is illustrated by an analogy between reader-writer locking and RCU. This analogy is shown in Table 3.1 and expanded on in Section 5.2.5 on Page 153, illustrating this common use of RCU. The left-hand column of this table shows a reader-writer-locked linked-list search and delete function, while the right-hand column shows the RCU equivalents. Figures 3.7 and 3.8 depict common code used by both the reader-writer locking and the RCU variants.

The locking in the left-hand side of Table 3.1 ensures that any deletions are atomic from the viewpoint of the searching code. However, if the list is read-intensive, the overhead of the locks in the `search()` code can be excessive, with the resulting contention restricting scaling.

RCU permits the read-side locking to be eliminated in non-preemptive environments, as shown in the upper right portion of the table. Write-side locking can then be converted to the less-expensive spinlocks, as shown on the lower right portion of the table. However, the call to `free()` must be deferred through use of the `call_rcu()` primitive.

```
 1 struct el {
 2   struct el *next;
 3   struct el *prev;
 4   long key;
 5   spinlock_t mutex;
 6   struct rcu_head *rcu;
 7   int data;
 8   /* Other data fields */
 9 };
10 spinlock_t listmutex;
11 struct el head;
```

Figure 3.7: Search-List Data Structures

```
 1 struct el *_search(long key)
 2 {
 3   struct el *p;
 4
 5   p = head.next;
 6   while (p != &head) {
 7     if (p->key == key) {
 8       return (p);
 9     }
10     p = p->next;
11   }
12   return (NULL);
13 }
```

Figure 3.8: Internal Search Algorithm

Table 3.1: Reader-Writer Locking and RCU

Reader-Writer Lock

```
 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   read_lock(&listmutex);
 6   p = _search(key);
 7   if (p != NULL)
 8     *result = p->data;
 9   read_unlock(&listmutex);
10   return (p != NULL);
11 }
```

RCU

```
1 int search(long key, int result)
2 {
3   struct el *p;
4   p = _search(key);
5   if (p != NULL)
6     *result = p->data;
7   return (p != NULL);
8 }
```

Reader-Writer Lock

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listmutex);
 6   p = _search(key);
 7   if (p == NULL) {
 8     write_unlock(&listmutex);
 9   } else {
10     p->next->prev = p->prev;
11     p->prev->next = p->next;
12     spin_unlock(&p->mutex);
13     write_unlock(&listmutex);
14     free(p);
15   }
16   return (p != NULL);
17 }
```

RCU

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   p = _search(key);
 7   if (p == NULL) {
 8     spin_unlock(&listmutex);
 9   } else {
10     p->next->prev = p->prev;
11     p->prev->next = p->next;
12     spin_unlock(&p->mutex);
13     spin_unlock(&listmutex);
14     call_rcu(&p->rcu, free, p);
15   }
16   return (p != NULL);
17 }
```

The `call_rcu()` primitive enqueues a callback that, after a grace period has elapsed, invokes its second argument, passing this function its third argument. So, in this example, `call_rcu()` will enqueue a callback that invokes `free(p)` after a grace period elapses. The `call_rcu()` function uses its first argument to track this callback through the duration of the grace period.

Because readers run concurrently with writers, updates of multiple fields must be handled carefully. A number of approaches are discussed in Section 5.3 on Page 159.

3.4.2 RCU Use of Deferred Destruction

A key property of RCU is deferral of destruction until all currently active read-side operations complete.

This deferred destruction process is illustrated by Figure 3.9. Part (1) of this figure shows the initial state of the list with each element possibly referenced by concurrent invocations of `search()`. Part (2) of the figure shows the list after the `delete()` function has removed element B from the list. At this point, searches in the forward direction that have arrived at element C may or may not have passed through element B, as indicated by the shaded arrows emanating from element A. Similarly, searches in the reverse direction that have arrived at element A also may or may not have passed through element B, as indicated by the shaded arrows emanating from element C. However, subsequent searches cannot obtain a new reference to element B. Part (3) shows the state after a grace period has elapsed, by which time there can no longer be any searches referencing element B. Part (4) shows the state after `call_rcu()` invokes `free(p)`, sending element B back to the free-memory pool.

This process requires a mechanism to compute the duration of a grace period, which was supplied by McKenney and Slingwine [81, 108], and independently by Gamsa et al. [30] and by Hennessy, Osisek, and Seigh [39]. These papers reported on work done much earlier, indeed, RCU was running in production in datacenters by 1993 on DYNIX/ptx and a similar technique was in use in the mid-80s on IBM's VM/XA [39].

RCU is a valuable technique, but it does not provide readers any guarantee of consistency (beyond that guaranteed by the hardware) or of freshness of data. RCU inherently

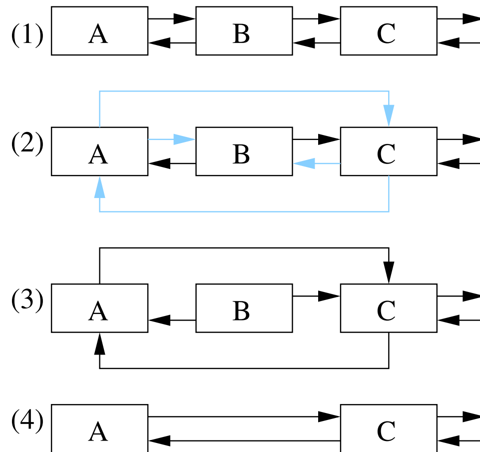


Figure 3.9: RCU Destruction Determination

cannot provide such guarantees, since readers do not coordinate directly with writers, and can therefore see old versions of the data (stale data), and can in fact see different versions of the data on consecutive accesses (inconsistent data). However, in a surprising number of situations, such guarantees are not needed, and in many other situations, the algorithm may be transformed to remove its reliance on consistency and freshness, as is described in Section 5.3 on Page 159.

3.5 RCU Analogies and Design Patterns

As noted earlier, RCU permits partial ordering of accesses and updates to such an extent that RCU readers can see stale and inconsistent data. RCU readers see stale data when they race with an update, and continue referencing a now-old version. RCU readers see inconsistent data in cases where writers perform non-atomic updates, such as tree-rebalancing operations. Interestingly enough, it turns out that many algorithms can tolerate staleness and inconsistency.

For example, consider a TCP/IP routing table implemented as a hash table with singly linked hash chains. Staleness is not a problem because TCP/IP routing protocols have substantial built-in delays, in some cases, measured in minutes, in order to prevent routing instabilities from being induced by transient failures. These delays mean that routing data

can be stale before it even arrives at the TCP/IP host. Therefore, small additional delays due to RCU pose no additional problem. In either case, the upper level TCP/IP protocol will retransmit any data that continues to be sent down the wrong path.

Inconsistency is not a problem given that the route-table search procedure traverses the list in a single direction. Again, any packets dropped due to a routing entry being missed by an RCU reader will be retransmitted by upper-level protocols.

However, the TCP/IP protocol suite was specifically designed to be robust in a dynamic network. Not all algorithms are designed to operate correctly when under active attack, and thus not all algorithms gracefully tolerate stale and inconsistent data. Such intolerance is one of the basic research challenges presented by RCU.

Therefore, one of the aims of this dissertation is to identify design patterns that transform such algorithms into functionally equivalent algorithms that are capable of tolerating stale and inconsistent data. Such a set of transformational design patterns is presented in Section 5.3 on Page 159. These transformational design patterns move RCU from the realm of ad-hoc concurrency hacks to that of generally applicable synchronization mechanisms.

3.6 Discussion

RCU makes use of deferred destruction, so the relationship between RCU and prior work is as shown in Table 2.2 on Page 69.

There are some strong relationships between RCU and other synchronization mechanisms. For example, it resembles reader/writer locking and brlock in permitting concurrent reads and avoiding read-side contention; it resembles brlock and semantic-based methods in being an asymmetric mechanism; and it resembles data locking, partitioning, and non-blocking synchronization in reducing write-side contention. It very closely resembles mechanisms based on deferred destruction, as is to be expected.

However, RCU is distinguished from all synchronization mechanism other than those involving deferred destruction by

1. total avoidance of memory latency, pipeline stalls, locking, and contention in read-side critical sections, and

2. permitting concurrent reading and writing in all cases.

Although data locking, partitioning, non-blocking synchronization, and semantics-based methods do permit some reader/writer concurrency, they do so in only a limited fashion. Data locking and partitioning permit concurrency only among operations affecting different objects. Non-blocking synchronization causes conflicting reads or writes to fail: despite the fact that the operations run in parallel, only one such conflicting operation is permitted to succeed. Some semantics-based methods do permit full concurrency, the split counter discussed in Section 2.2.13 on Page 41 being one such example. However, transaction-based methods must by their nature reduce concurrency in some cases [140]. Nevertheless, RCU draws heavily on the concepts embodied by many of these earlier synchronization mechanisms.

Since RCU is closely related to a number of the deferred-destruction mechanisms, it is worthwhile to examine them more closely, as shown in Table 3.2. The meanings of the entries are similar to those for Table 2.2 on Page 69:

A Avoids the specified type of overhead.

P Parallelizes the specified type of overhead.

Y Supports the specified attribute.

N Includes special support for NUMA machines.

S Includes special support for SMMP machines.

Again, a lower-case character indicates partial support.

These deferred-destruction mechanisms were covered in Section 2.2.20 on Page 62; this table summarizes that discussion.

The major distinction between the work presented in this dissertation and prior mechanisms based on deferred destruction is the design patterns and transformational design

Table 3.2: Attributes of Deferred Destruction Mechanisms

	Year of Publication	Year in Production	Architecture Optimizations	Appropriate for OS Kernel	Reads:				Writes:			
					Pipeline Flushes	Memory Latency	Contention	Locking	Pipeline Flushes	Memory Latency	Contention	Locking
Kung/Lehman	80						A	A	?	?	?	?
Manber/Ladner	82				A	A	A	A	?	?	?	?
Passive Serialization	89	86?		Y	A	A	A	A	p	p	a	p
Pugh	90			y		?	A	A	p	p	a	p
Chaotic Relaxation	91				a	a	a	a	a	a	a	a
Jacobson	93				A	A	A	A	?	?	?	?
John	95				A	A	A	A	?	?	?	?
Pu	95			Y		a	a	a				
Dynix/PTX RCU	98	93	SN	Y	A	A	A	A	p	p	a	p
K42 Generations	99		SN	Y	A	A	A	A	p	p	a	p
NBS & Hazard Pointers	02			y		a	a	A	p	p	a	A

patterns introduced by this dissertation. As noted earlier, the design patterns are introduced into an existing locking design pattern language and the transformational design patterns are presented in Chapter 5. Prior to that, Chapter 4 describes a variety of RCU implementations in more detail.

Chapter 4

Implementing RCU

RCU has been implemented in DYNIX/ptx [81, 108], SuSE 7.3 Update, Linux 2.6 [11, 121], K42/Tornado [30], and VM/XA [39]. This chapter describes the K42/Tornado and several Linux implementations of the RCU infrastructure; uses of this infrastructure are described in Chapter 5.

This chapter presents the Linux 2.6 kernel's RCU API in Section 4.1. It then presents a number of different possible implementations of RCU infrastructure for the Linux 2.6 kernel and for K42 as well in Section 4.2. Finally, it compares and contrasts these implementations in Section 4.6.

This author's role in the work described in this chapter was that of architect, design reviewer, and code reviewer for the Linux implementations, and co-inventor and implementor for the DYNIX/ptx implementation [81]. This author also did the comparative analysis of the algorithms. The people who did the actual design and coding of the various Linux algorithms are called out in the individual sections.

A key innovation required in porting RCU to Linux was enabling many different RCU techniques to be used on many different computer architectures. Earlier work had been restricted to either using a single RCU technique on a single computer architecture (e.g., VM/XA on the IBM 370 architecture), using many different RCU techniques on a single computer architecture (e.g., DYNIX/ptx on the x86 architecture), or using a single RCU technique on a small number of computer architectures (e.g., Tornado or K42 on PowerPC, MIPS, and Opteron).

```
void rcu_read_lock(void);
void rcu_read_unlock(void);
void synchronize_kernel(void);
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

Figure 4.1: RCU API

4.1 RCU API

The RCU API defines the set of services provided by the RCU infrastructure. Understanding this set of services, described in this section, is key to understanding RCU. However, it is also necessary to understand how best to use these services, which is taken up in Chapter 5 on Page 137.

An explicit RCU API also enables experimentation with different implementations of the RCU infrastructure. As long as a given patch faithfully meets the RCU API, it may be easily dropped into the Linux kernel for testing.

The diverse RCU implementations are hidden from the in-kernel RCU user by a well-defined API, shown in Figure 4.1. This API makes heavy use of the Linux `list_head` doubly linked list header. Many people were involved in the discussions that defined this API, particularly Dipankar Sarma, Rusty Russell, and Andrea Arcangeli. Note that although this API is current as of the Linux 2.6.0 kernel, it will likely be subject to change.

Section 4.1.1 describes the core RCU API used by readers, Section 4.1.2 describes the core RCU API used by writers, and Section 4.1.3 describes the RCU extensions to Linux's list-manipulation API. The combined read- and write-side RCU API is shown in Figure 4.1, and the list-manipulation RCU API is shown in Figure 4.2.

4.1.1 Read-Side RCU API

The read-side RCU API is used by readers to demark the extent of a read-side RCU critical section using the `rcu_read_lock()` and `rcu_read_unlock()` primitives described below.

```
list_add_rcu(struct list_head *new, struct list_head *head);
list_add_tail_rcu(struct list_head *new, struct list_head *head);
list_del_rcu(struct list_head *entry);
list_for_each_rcu(struct list_head *pos, struct list_head *head)
list_for_each_safe_rcu(struct list_head *pos, struct list_head *n, struct list_head *head)
list_for_each_entry_rcu(struct list_head *pos, struct list_head *head, struct list_head *member)
list_for_each_continue_rcu(struct list_head *pos, struct list_head *head)
hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);
hlist_del_rcu(struct hlist_node *n);
```

Figure 4.2: RCU List API

```
void rcu_read_lock(void);
```

The `rcu_read_lock()` primitive marks the beginning of a read-side RCU critical section. Read-side RCU critical sections may be nested. In preemptive Linux kernels, it suppresses preemption, while in non-preemptive Linux kernels, it can be a no-op. It is possible to construct RCU mechanisms which do not require preemption suppression, even in non-preemptive Linux kernels, as described in Section C.3 on Page 343. This approach was pioneered by the K42 research operating system, as described in Section 4.4.3 on Page 130.

In either case, nothing prevents readers from being affected by concurrent updates. Although there are some algorithms that tolerate this situation, there are many algorithms that do not. See Section 5.3 on Page 159 for a discussion of ways of transforming such algorithms into forms that tolerate updaters interfering with readers.

```
void rcu_read_unlock(void);
```

The `rcu_read_unlock()` primitive marks the end of a read-side RCU critical section. As such, in non-preemptive Linux kernels, it can again be a no-op, while in preemptive Linux kernels, it re-enables preemption. Since read-side RCU critical sections may be nested, this re-enabling must check for such nesting.

Note also that the `rcu_read_unlock()` primitive is a good place to put debug checks, for example, checks that verify that the read-side RCU critical section did not pass through any quiescent states.

Perhaps the most important function of the `rcu_read_lock()` and `rcu_read_unlock()` primitives is to document the extent of the RCU read-side critical section, thus making

```
 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   rcu_read_lock();
 6   p = head.next;
 7   while (p != &head) {
 8     if (p->key == key) {
 9       *result = p->data;
10       rcu_read_unlock();
11       return (1);
12     }
13     p = p->next;
14   }
15   rcu_read_unlock();
16   return (0);
17 }
```

Figure 4.3: Example Read-Side RCU Critical Section

it less likely for a programmer to inadvertently insert an blocking operation, such as a semaphore acquisition.

Example Use and Discussion

The `rcu_read_lock()` and `rcu_read_unlock()` primitives are used to bracket a read-side RCU critical section, such as the search code shown in Figure 4.3. This code is similar to that in Figure 3.8 on Page 91, but with the read-side RCU primitives inserted.

4.1.2 Write-Side RCU API

The write-side RCU primitives allow the caller to defer an action until after all pre-existing read-side RCU critical sections have completed execution. There are two variants, the `synchronize_kernel()` primitive which blocks for a grace period, and the `call_rcu()` primitive which returns immediately, having arranged for deferred invocation of a specified function. These primitives are described in the following sections.

```
void synchronize_kernel(void);
```

The `synchronize_kernel()` primitive blocks until the end of a full grace period. This is an easy-to-use primitive, since upon its return, all pre-existing read-side RCU critical sections will have completed execution. It also allows concurrent callers to share a grace

period. However, it incurs expensive context-switch overhead and cannot be called with locks held or from interrupt handlers.

```
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

In contrast, `call_rcu()` schedules invocation of the callback function `func(arg)` after a full grace period, and may thus be thought of as a way to invoke `func(arg)` once all pre-existing read-side RCU critical sections have completed execution. Since `call_rcu()` never sleeps, it may be called with locks held and from interrupt context. The `call_rcu()` function uses its `struct rcu_head` argument to remember its callback function and argument during the grace period. An `rcu_head` is often placed within the structure being protected by RCU, eliminating the need to separately allocate it.

Note that the `synchronize_kernel()` primitive is actually implemented in terms of `call_rcu()`.

Example Use and Discussion

Both `call_rcu()` and `synchronize_kernel()` permit destructive operations to be deferred until all pre-existing read-side RCU critical sections complete execution. The `synchronize_kernel()` primitive may be used inline by code that can safely block, as shown in Figure 4.4. Here, the `synchronize_kernel()` on line 14 prevents the `kfree()` on line 15 from executing until all pre-existing read-side critical sections have completed execution. Only such pre-existing critical sections can possibly hold a reference to the element being deleted, so once they complete, the element may safely be freed.

In contrast, Figure 4.5 shows code that may be called in cases where blocking is not permitted, such as from interrupt handlers, spinlock critical sections, and code holding references to CPU-specific variables. Here, line 14 invokes `call_rcu()`, which causes

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   p = search(key);
 7   if (p == NULL) {
 8     spin_unlock(&listmutex);
 9   } else {
10     p->next->prev = p->prev;
11     p->prev->next = p->next;
12     spin_unlock(&p->mutex);
13     spin_unlock(&listmutex);
14     synchronize_kernel();
15     kfree(p);
16   }
17   return (p != NULL);
18 }
```

Figure 4.4: Example Blocking Write-Side RCU Critical Section

`free(p)` to be invoked after all pre-existing read-side RCU critical sections complete execution.

Experience has shown that the `synchronize_kernel()` style is simpler to use, so it should be preferred where calling-context and performance considerations permit. However, if the deletion must be done from a context such as a spinlock critical section or an interrupt handler, or if the deletion operation is performance-critical, then the `call_rcu()` form must be used instead.

In the Linux 2.6.6 kernel, there are 32 uses of `synchronize_kernel()` (including the synonym `synchronize_net()`) and 12 uses of `call_rcu()`.

4.1.3 List-Manipulation RCU API

Most modern microprocessors feature weak memory-consistency models, which require special memory-barrier instructions, which are discussed in Section 2.2.10 on Page 30. These memory-barrier instructions must be used when manipulating data structures accessed by synchronization-free readers in order to ensure that the readers see a consistent view of the data structure. A few CPUs require additional memory-barrier instructions for readers, as described in Appendix B on Page 322. Both the read- and write-side memory-barrier instructions are difficult to understand and even more difficult to test,

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   p = search(key);
 7   if (p == NULL) {
 8     spin_unlock(&listmutex);
 9   } else {
10     p->next->prev = p->prev;
11     p->prev->next = p->next;
12     spin_unlock(&p->mutex);
13     spin_unlock(&listmutex);
14     call_rcu(&p->rcu, kfree, p);
15   }
16   return (p != NULL);
17 }
```

Figure 4.5: Example Non-Blocking Write-Side RCU Critical Section

so this author extended the Linux list-manipulation API to include memory barriers, as suggested by Manfred Spraul [114] and as shown in Figure 4.2. Dipankar Sarma took over this patch, and it was later accepted into the Linux kernel [122].

Doubly Linked List Addition

```
void list_add_rcu(struct list_head *new, struct list_head *head);
void list_add_tail_rcu(struct list_head *new, struct list_head *head);
```

The `list_add_rcu()` and `list_add_tail_rcu()` primitives insert an element at the head and tail, respectively, of a doubly linked list, using whatever memory-barrier instructions are necessary to ensure that the pointers within the new object are committed to memory before committing the pointers referencing the new object.

Doubly Linked List Deletion

```
void list_del_rcu(struct list_head *entry);
```

The `list_del_rcu()` primitive removes the specified element from its linked list. Unlike the `list_del()` primitive, the pointers in the newly removed entry are not “poisoned”, or set to an invalid value, since doing so would cause concurrent readers to fail.

Doubly Linked List Traversal

```
void list_for_each_rcu(struct list_head *pos, struct list_head *head);
void list_for_each_safe_rcu(struct list_head *pos,
                            struct list_head *n,
                            struct list_head *head);
void list_for_each_entry_rcu(struct list_head *pos,
                             struct list_head *head,
                             struct list_head *member);
void list_for_each_continue_rcu(struct list_head *pos,
                                struct list_head *head);
```

The `list_for_each_rcu()` primitive and its variants traverse the specified RCU-protected linked list, expanding into the corresponding C "for" loop. The "safe" variant permits deletion of the current element, the "entry" variant provides a pointer to the enclosing data structure rather than the enclosed `struct list_head`, and the "continue" variant permits picking up where a previous `list_for_each()`-style primitive left off.

Singly Linked List Manipulation

The `hlist` doubly linked list primitives were recently added by Andi Kleen in order to reduce memory requirements for large hash tables. Although the `hlist` is doubly linked, the header consists of but a single pointer. This smaller header permits `hlist` hash table arrays to be half the size of those for normal doubly-linked lists.

```
void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h);
```

The `hlist_add_head_rcu()` primitive adds the specified entry at the head of the specified `hlist`, executing any memory-barrier instructions needed to ensure that other CPUs see the new entry's pointers before any pointers are set to reference the new entry. As with `hlist_add_head_rcu()`, these memory barriers are required to permit readers to concurrently search the list without use of synchronization instructions.

```
void hlist_del_rcu(struct hlist_node *n);
```

```
 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   rcu_read_lock();
 6   p = head.next;
 7   list_for_each_entry_rcu(p, head, list) {
 8     if (p->key == key) {
 9       *result = p->data;
10       rcu_read_unlock();
11       return (1);
12     }
13   }
14   rcu_read_unlock();
15   return (0);
16 }
```

Figure 4.6: Example Read-Side Use of RCU List-Manipulation Primitives

The `hlist_del_rcu()` primitive removes an element, but differs from `hlist_del()` in that it refrains from poisoning the "next" list pointer, thus allowing concurrent readers to proceed without failure.

List-Manipulation RCU API Examples

Figure 4.6 shows how the `list_for_each_entry_rcu()` primitive may be used. It replaces line 7 and deletes line 13 of Figure 4.3, simultaneously simplifying the code and adding any required memory-barrier instructions.

Figure 4.7 shows how the `list_del_rcu()` primitive may be used. It replaces the two pointer-manipulation statements on lines 10 and 11 of Figure 4.4, simplifying the code as well as adding any required memory-barrier instructions.

The other RCU list-manipulation APIs are used in a similar fashion. These APIs have proven extremely useful, which is not surprising, given that experience with the raw memory-barrier primitives has shown them to be extremely difficult to use, test, and debug.

4.2 Implementing Grace-Period Detection

The following sections summarize several different types of `call_rcu()` implementation. The two basic SMMP approaches are (1) inducing quiescent states, which is covered in

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   p = search(key);
 7   if (p == NULL) {
 8     spin_unlock(&listmutex);
 9   } else {
10     list_del_rcu(p);
11     spin_unlock(&p->mutex);
12     spin_unlock(&listmutex);
13     synchronize_kernel();
14     free(p);
15   }
16   return (p != NULL);
17 }
```

Figure 4.7: Example Write-Side Use of RCU List-Manipulation Primitives

Section 4.3, and (2) observing naturally occuring quiescent states, which is covered in Section 4.4. Section 4.5 discusses special considerations for uniprocessors.

4.3 Inducing Quiescent States

The simplest way to detect a grace period is to create one when needed. Figure 4.8 shows how such an RCU implementation operates. The boxes represent non-preemptible kernel execution, the space between them represents candidate quiescent states (e.g., context switch, user mode, idle loop, or user-mode execution), and each numbered arrow represents a thread, with time progressing to the right.

The leftmost dashed line indicates the time of the first phase of the RCU (e.g., removal of an element from a list). The second phase of the update (e.g., the actual freeing of the element) may proceed as soon as all operations that were in progress during the first phase have completed, namely, operations A, E, and L. The earliest time the second phase can safely be initiated is indicated by the rightmost dashed line in Figure 4.8, and the distance between the two dashed lines is the minimum allowable duration of the grace period, since during this time, there may exist threads that still hold references to the list element.

As noted in Section 3.2.3 on Page 78, in a non-preemptive kernel, it is sufficient to force a context switch on each CPU, since any threads that are not running at the time of the

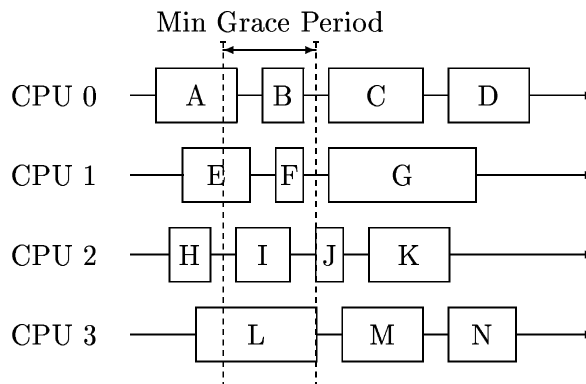


Figure 4.8: RCU Grace Period

update are by definition in an extended quiescent state. Therefore, a simple procedure to determine when the second phase may safely be initiated in a non-preemptive operating-system kernel is depicted in Figure 4.9. The updater simply forces itself to execute on each CPU in turn. This is safe because the non-preemptive nature of the kernel will delay the updater's execution until all operations in progress on each CPU have completed. The boxes labeled "u" represent this updater's execution. Once it has run on each CPU, then all operations that were in progress during phase one must have completed. Note that this procedure is a way to quickly *detect* the end of a grace period; it in no way causes any of the ongoing operations to complete sooner than they would have otherwise.[1]

4.3.1 Simple Induced Quiescent States

Rusty Russell's `synchronize_kernel()` primitive shown in Figure 4.10, is a simple example of inducing quiescent states. This primitive was originally called `wait_for_rcu()`.

Lines 7 through 13 save the current scheduling state, and set up a FIFO scheduling policy with sufficient priority to preempt any tasks that are running in user mode and also any tasks running in the kernel with preemption enabled. Lines 15 and 16 create a mask that allows the task to run on any CPU. The loop on lines 19 through 22 repeatedly eliminates the current CPU from the set allowed to run this task, then yields the CPU.

[1]In fact, the additional scheuduler execution may in fact delay these ongoing operations somewhat.

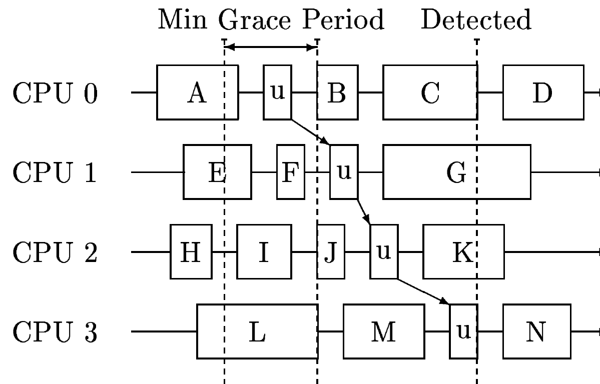


Figure 4.9: Simple Grace-Period Detection

```
 1 void synchronize_kernel(void)
 2 {
 3    unsigned long cpus_allowed;
 4    unsigned long policy;
 5    unsigned long rt_priority;
 6    /* Save current state */
 7    cpus_allowed = current->cpus_allowed;
 8    policy = current->policy;
 9    rt_priority = current->rt_priority;
10    /* Create an unreal time task. */
11    current->policy = SCHED_FIFO;
12    current->rt_priority = 1001 +
13       sys_sched_get_priority_max(SCHED_FIFO);
14    /* Make us schedulable on all CPUs. */
15    current->cpus_allowed =
16               (1UL<<smp_num_cpus)-1;
17
18    /* Eliminate current cpu, reschedule */
19    while ((current->cpus_allowed &= ~(1 <<
20            cpu_number_map(
21             smp_processor_id())))  != 0)
22       schedule();
23    /* Back to normal. */
24    current->cpus_allowed = cpus_allowed;
25    current->policy = policy;
26    current->rt_priority = rt_priority;
27 }
```

Figure 4.10: Non-Preemptible Grace-Period Detection

Thus, upon loop completion, the task will have run on each of the CPUs, which means that each CPU will have completed whatever it was doing at the time of the call to `synchronize_kernel()`. It would therefore now be safe to free up an element that was unlinked from its list before the call to `synchronize_kernel()`. Lines 24 through 26 restore the scheduling state.

This code is quite straightforward, but this approach does have a few shortcomings:

1. it would not work in a preemptible kernel unless preemption is suppressed in all read-side critical sections,

2. it cannot be called from an interrupt handler because the `schedule()` call blocks, however, `schedule_task()` can be used to call it indirectly,

3. it cannot be called while holding a spinlock or with interrupts disabled (again because `schedule()` blocks), however, again, `schedule_task()` can be used to call it indirectly, and

4. `schedule()`'s per-CPU context switches incur substantial overhead.

Section 4.4.3 on Page 130 and Appendix C.3 on Page 343 describe two possible ways of addressing item 1. The following section describes the `call_rcu()` primitive that addresses items 2 and 3. Section 4.4.1 on Page 113 describes a faster grace-period-detection algorithm for non-preemptible read-side critical sections that addresses items 2, 3, and 4.

4.3.2 Induced Quiescent States With Batching (batch)

Another way of waiting for grace periods while holding spinlocks or from an interrupt handler is to define a `call_rcu()` function that queues callbacks onto a list. A separate `free_pending_rcus()` function can then invoke all the pending callbacks after forcing a grace period. This can also have the beneficial effect of amortizing the grace-period-detection overhead over multiple updates, greatly reducing the per-update overhead.

Figure 4.11 shows a straightforward implementation of these two functions. Note that `free_pending_rcus()` must be invoked from time to time. Appendix C.1 on Page 326 describes one way of accomplishing this within the Linux kernel.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *head))
 3 {
 4    unsigned long flags;
 5
 6    head->destructor = func;
 7    spin_lock_irqsave(&rcu_lock, flags);
 8    head->next = rcu_list;
 9    rcu_list = head;
10    spin_unlock_irqrestore(&rcu_lock, flags);
11 }
12
13 void free_pending_rcus(void)
14 {
15    struct rcu_head *list;
16
17    spin_lock_irq(&rcu_lock, flags);
18    list = rcu_list;
19    rcu_list = NULL;
20    spin_unlock_irq(&rcu_lock, flags);
21
22    /* If list nonempty, wait and destroy. */
23    if (list) {
24       synchronize_kernel();
25       while (list) {
26          struct rcu_head *next = list->next;
27
28          list->destructor(list);
29          list = next;
30       }
31    }
32 }
```

Figure 4.11: Non-Blocking Grace-Period Detection

Although batching the RCU callbacks via the `rcu_list` reduces overhead, the basic grace-period detection overhead is still quite high, and increases with increasing numbers of CPUs. The reason for this is that context switches are quite expensive. Given that context switches occur quite frequently during normal system operation, it is reasonable to ask whether it is possible to do better by observing and leveraging these naturally occurring context switches. Of course, waiting for naturally occuring context switches could potentially delay the detection of the end of the grace period, but this may not be a problem if this delay is sufficiently short.

4.4 Observing Naturally Occurring Quiescent States

When observing naturally occuring quiescent states, it is important to avoid significantly increasing their overhead. It is also important to determine efficiently, from the observations of quiescent states, when a given grace period has ended. The required efficiency is typically achieved by instrumenting the observed quiescent states with per-CPU counters, and feeding the results into a barrier computation. Once all CPUs have checked into the barrier, the grace period has ended. However, the barrier computation and the observation can be combined, as they are in the implementations described in Sections 4.4.2 and 4.4.3. As noted in Section 4.3.2, callbacks are used to permit RCU to be invoked from interrupt handlers or with spinlocks held, and also to promote efficiency via batching of callbacks, so that a single grace period may serve for a large number of updates.

4.4.1 Counters and Barrier (rcu-ltimer)

The *rcu-ltimer* implementation, designed and coded by Dipankar Sarma, is similar to the DYNIX/ptx implementation [108]. It inserts per-CPU-counter increments into the scheduler, and drives the barrier computation from the per-CPU timer interrupt handler. In the Linux 2.4 kernel, the per-CPU timer interrupt handler was architecture specific, requiring separate changes for each and every architecture that Linux supports. Fortunately, however, the advent of the `scheduler_tick()` function in the 2.6 kernel now permits architecture-independent timer processing. This permits counting user-mode execution as

a quiescent state, in addition to the idle loop and context switch. This section describes the 2.6 kernel implementation.

Overview of Implementation

The *rcu-ltimer* implementation uses callbacks and a grace-period barrier computation.

Callback Handling The callback handling is as shown in Figure 4.12. In accordance with the principles identified in Section 2.3.2 on Page 65, this implementation attempts to minimize cacheline transfers. One important way of accomplishing this is to allow the CPUs to manage their own callbacks, using the concept of data ownership to partition callback handling over the CPUs without requiring CPUs to acquire locks to access their own callbacks.

However, such partitioning means that CPUs may learn of the start or end of a grace period at different times, so that CPU 0 might start a new grace period before CPU 1 is aware that the old grace period has completed. CPU 1 must therefore be able to recognize that the old grace period has completed after the new one has started. This is accomplished by numbering the grace periods, with each CPU maintaining a private counter named `RCU_batch(cpu)` which tracks the number of the grace period that it is waiting on. A CPU can then compare its `RCU_batch(cpu)` against the global `rcu_ctrlblk.curbatch` counter to determine whether if these two counters do not match, the grace period that this CPU was waiting on has completed.

Callbacks waiting for the `RCU_batch(cpu)`th grace period are kept in `RCU_curlist(cpu)`. These callbacks will be invoked after this grace period elapses. However, new callbacks cannot be placed on this list, since the corresponding grace period has already started. Because callbacks must wait for a *full* grace period, new callbacks are instead placed on the `RCU_nxtlist(cpu)`, as indicated by the arrow from `call_rcu()` in the figure. When this CPU becomes aware that the `RCU_batch(cpu)`th grace period has ended, it invokes the callbacks in its `RCU_curlist(cpu)` list, then moves any callbacks from its `RCU_nxtlist(cpu)` to its `RCU_curlist(cpu)`.

If there were any callbacks to move, CPU must now update its `RCU_batch(cpu)` counter

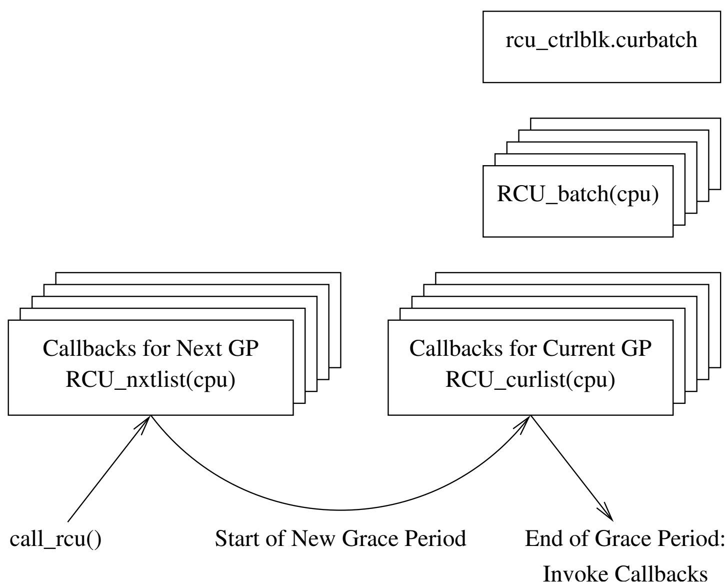


Figure 4.12: RCU Callback Tracking (rcu-ltimer)

to reflect the new grace period that is to be waited on. However, it cannot use the current value of the `rcu_ctrlblk.curbatch` counter, since that value corresponds to a grace period that has already started, and therefore might have started before the last callback was enqueued. Instead, the `RCU_batch(cpu)` counter must be set to one greater than the `rcu_ctrlblk.curbatch` counter, so that the callbacks will be invoked only after a full grace period has elapsed.

Grace-Period Barrier Computation This grace-period barrier computation relies on instrumenting the task scheduler and the per-CPU timer interrupt handler to detect quiescent states. The scheduler detects context switches, and the per-CPU timer interrupt handler detects user-mode execution. The per-CPU timer interrupt handler has a hook that observes the quiescent-state data and drives the barrier computation forward. As shown in Figure 4.13, a bit vector is used to keep track of which CPUs have passed through at least one quiescent state in the current grace period. When a grace period begins, each CPU's bit is set.

There are a pair of per-CPU counters that track quiescent states executed by this CPU,

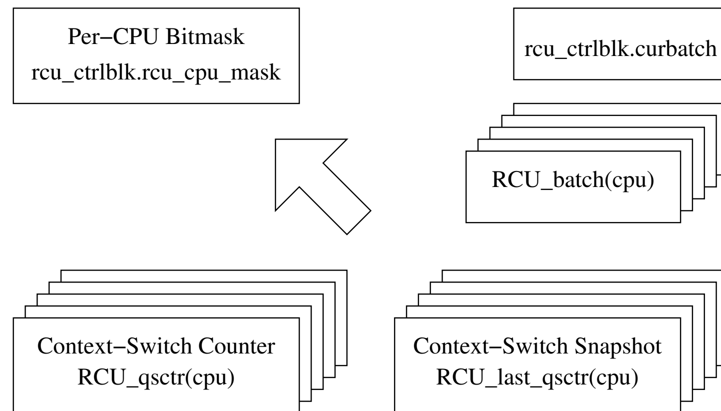


Figure 4.13: *rcu-ltimer* Barrier-Computation Variables

which are also shown in the figure. The `RCU_qsctr(cpu)` variable is incremented by the scheduler on each context switch, and the `RCU_last_qsctr(cpu)` variable is used to snapshot the value of `RCU_qsctr(cpu)` once the corresponding CPU realizes that a new grace period has started, due to its `RCU_batch(cpu)` differing from `rcu_ctrlblk.curbatch`. Since the next pass through the scheduler will increment the `RCU_qsctr(cpu)` variable, as soon as the CPU notes that the values of its two counters are different, it clears its bit in the bitmask, as indicated by the block arrow in the figure. When all CPUs' bits have been cleared, the grace period has ended.

Code Walkthrough

The implementation contains code to:

1. register RCU callbacks,
2. count context switches, and
3. track and invoke RCU callbacks, and
4. perform the grace-period barrier computation.

Register RCU Callbacks To register an RCU callback, the `call_rcu()` function constructs a callback and enqueues it onto a per-CPU `RCU_nxtlist`, as shown in Figure 4.14.

Note that lines 8 and 11 mask interrupts to prevent interrupt handlers from concurrently accessing the list.

```
 1 void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg)
 2 {
 3   int cpu;
 4   unsigned long flags;
 5
 6   head->func = func;
 7   head->arg = arg;
 8   local_irq_save(flags);
 9   cpu = smp_processor_id();
10   list_add_tail(&head->list, &RCU_nxtlist(cpu));
11   local_irq_restore(flags);
12 }
```

Figure 4.14: *rcu-ltimer* call_rcu() Implementation

Count Context Switches The scheduler is instrumented by inserting a simple counter increment, as shown on line 4 of Figure 4.15. The `task_cpu(prev)` returns the CPU that was running the task being context-switched away from.

```
 1 switch_tasks:
 2   prefetch(next);
 3   clear_tsk_need_resched(prev);
 4   RCU_qsctr(task_cpu(prev))++;
 5
 6   prev->sleep_avg -= run_time;
 7   if ((long)prev->sleep_avg <= 0){
 8     prev->sleep_avg = 0;
 9     if (!(HIGH_CREDIT(prev) || LOW_CREDIT(prev)))
10       prev->interactive_credit--;
11   }
12   prev->timestamp = now;
```

Figure 4.15: *rcu-ltimer* Scheduler Instrumentation

Track and Invoke RCU Callbacks Lines 8 and 9 of Figure 4.16 show the RCU hook in the per-CPU timer interrupt handler. If there is at least one RCU callback pending that needs this CPU's attention, `rcu_check_callbacks()` is invoked to advance the grace-period barrier computation, if possible.

Figure 4.17 shows the code that checks to see if there are any RCU callbacks pending that require this CPU's attention, returning 1 if so and 0 otherwise. Lines 3-6 check

```
 1 void scheduler_tick(int user_ticks, int sys_ticks)
 2 {
 3   int cpu = smp_processor_id();
 4   struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
 5   runqueue_t *rq = this_rq();
 6   task_t *p = current;
 7
 8   if (rcu_pending(cpu))
 9     rcu_check_callbacks(cpu, user_ticks);
10
11   /* note: this timer irq context must be accounted for as well */
12   if (hardirq_count() - HARDIRQ_OFFSET) {
```

Figure 4.16: *rcu-ltimer* Per-CPU Timer Instrumentation

for callbacks on this CPU needing attention: Lines 3 and 4 check for the expiration of the grace period corresponding to the RCU callbacks in this CPU's `RCU_curlist(cpu)` list, and Lines 5 and 6 check for new callbacks having been registered in this CPU's `RCU_nxtlist(cpu)` when there are no callbacks on this CPU waiting for the current grace period to expire on `RCU_curlist(cpu)`. Line 7 checks for callbacks on other CPU's needing attention, which is indicated by the current CPU's bit still being set in the global `rcu_ctrlblk.rcu_cpu_mask`. Note that all RCU callbacks need this CPU to pass through a quiescent state at least once during each grace period, but RCU callbacks that were registered on this CPU also need (1) to be advanced from `RCU_nxtlist(cpu)` to `RCU_curlist` and (2) to be invoked once the corresponding grace period expires.

```
 1 static inline int rcu_pending(int cpu)
 2 {
 3   if ((!list_empty(&RCU_curlist(cpu)) &&
 4        rcu_batch_before(RCU_batch(cpu), rcu_ctrlblk.curbatch)) ||
 5       (list_empty(&RCU_curlist(cpu)) &&
 6        !list_empty(&RCU_nxtlist(cpu))) ||
 7       cpu_isset(cpu, rcu_ctrlblk.rcu_cpu_mask))
 8     return 1;
 9   else
10     return 0;
11 }
```

Figure 4.17: *rcu-ltimer* RCU-Callback-Pending Check

If `scheduler_tick()` in Figure 4.16 finds that some pending RCU callbacks require this CPU's attention, it invokes `rcu_check_callbacks()`, which is shown in Figure 4.18. Lines 3-6 check for this CPU being in an extended quiescent state, in other words, if

```
1 void rcu_check_callbacks(int cpu, int user)
2 {
3   if (user ||
4       (idle_cpu(cpu) && !in_softirq() &&
5         hardirq_count() <= (1 << HARDIRQ_SHIFT)))
6     RCU_qsctr(cpu)++;
7   tasklet_schedule(&RCU_tasklet(cpu));
8 }
```

Figure 4.18: *rcu-ltimer* Tasklet Scheduling

the per-CPU timer interrupted user code, or if it interrupted the idle loop–as opposed to interrupting an interrupt that interrupted the idle loop. Regardless of whether this CPU is in an extended quiescent state, line 7 schedules `rcu_process_callbacks()` to be invoked from a tasklet upon return from interrupt.

The `rcu_process_callbacks()` function is shown in Figure 4.19, which (1) advances callbacks, (2) starts a new grace period, and (3) checks for this CPU having passed through a quiescent state. Lines 6-10 check to see if the grace period corresponding to the callbacks in `RCU_curlist(cpu)` has expired, and, if so, puts them in a local `list`. Line 12 disables interrupts in order to prevent their handlers from registering callbacks while the `RCU_nxtlist(cpu)` is being manipulated.

Line 13 checks to see if the `RCU_curlist(cpu)` is empty and the `RCU_nxtlist(cpu)` is not, which indicates that newly registered callbacks may be advanced to start waiting for a grace period. If so, lines 14-16 move the callbacks and re-enable interrupts, and lines 21-24 associate this batch of callbacks with the grace period immediately following the current one. The `rcu_ctrlblk.mutex` protects against CPUs concurrently starting new batches.

Line 28 invokes `rcu_check_quiescent_state()` in order to determine if this CPU has passed through a quiescent state. If there were any callbacks in `RCU_curlist(cpu)` whose grace period has expired, lines 29 and 30 call `rcu_do_batch()` in order to invoke them.

Figure 4.20 shows `rcu_do_batch()`, which invokes RCU callbacks whose grace period has expired. This function simply walks the list, deleting each entry in turn on line 8 and invoking the corresponding callback function on line 10. It is important that the deletion happen first, since the callback function will almost certainly free up the callback!

```
 1 static void rcu_process_callbacks(unsigned long unused)
 2 {
 3   int cpu = smp_processor_id();
 4   LIST_HEAD(list);
 5
 6   if (!list_empty(&RCU_curlist(cpu)) &&
 7       rcu_batch_after(rcu_ctrlblk.curbatch, RCU_batch(cpu))) {
 8     list_splice(&RCU_curlist(cpu), &list);
 9     INIT_LIST_HEAD(&RCU_curlist(cpu));
10   }
11
12   local_irq_disable();
13   if (!list_empty(&RCU_nxtlist(cpu)) && list_empty(&RCU_curlist(cpu))) {
14     list_splice(&RCU_nxtlist(cpu), &RCU_curlist(cpu));
15     INIT_LIST_HEAD(&RCU_nxtlist(cpu));
16     local_irq_enable();
17
18     /*
19      * start the next batch of callbacks
20      */
21     spin_lock(&rcu_ctrlblk.mutex);
22     RCU_batch(cpu) = rcu_ctrlblk.curbatch + 1;
23     rcu_start_batch(RCU_batch(cpu));
24     spin_unlock(&rcu_ctrlblk.mutex);
25   } else {
26     local_irq_enable();
27   }
28   rcu_check_quiescent_state();
29   if (!list_empty(&list))
30     rcu_do_batch(&list);
31 }
```

Figure 4.19: *rcu-ltimer* Callback Advancement

```
 1 static void rcu_do_batch(struct list_head *list)
 2 {
 3   struct list_head *entry;
 4   struct rcu_head *head;
 5
 6   while (!list_empty(list)) {
 7     entry = list->next;
 8     list_del(entry);
 9     head = list_entry(entry, struct rcu_head, list);
10     head->func(head->arg);
11   }
12 }
```

Figure 4.20: *rcu-ltimer* Callback Invocation

Grace-Period Barrier Computation The `rcu_start_batch()` function shown in Figure 4.21 starts a new grace period with the specified batch number. Lines 3-5 update `rcu_ctrlblk.maxbatch` if needed, the purpose of this being to allow RCU to determine when it can go idle. Lines 6-8 check to see if the requested grace period has already elapsed or if there is an ongoing grace period that precedes the desired one, returning if so, since in either case, it is not appropriate to start the desired grace period. Otherwise, line 10 sets all the bits in the per-CPU bitmap, indicating that each CPU needs to pass through a quiescent state to complete the newly started grace period.

```
 1 static void rcu_start_batch(long newbatch)
 2 {
 3   if (rcu_batch_before(rcu_ctrlblk.maxbatch, newbatch)) {
 4     rcu_ctrlblk.maxbatch = newbatch;
 5   }
 6   if (rcu_batch_before(rcu_ctrlblk.maxbatch, rcu_ctrlblk.curbatch) ||
 7       !cpus_empty(rcu_ctrlblk.rcu_cpu_mask)) {
 8     return;
 9   }
10   rcu_ctrlblk.rcu_cpu_mask = cpu_online_map;
11 }
```

Figure 4.21: *rcu-ltimer* Starting New Grace Period

The `rcu_check_quiescent_state()` function shown in Figure 4.22 checks to see if this CPU has passed through a quiescent state for the first time during the current grace period, and, if so, whether this marks the end of the current grace period. It also does end-of-grace-period processing and starts up the next grace period if required.

Lines 5 and 6 check to see if this CPU has already passed through a quiescent state during the current grace period, returning if so. Lines 13-16 check to see if this CPU is not yet aware of the current grace period, taking a snapshot of `RCU_qsctr(cpu)` in `RCU_last_qsctr(cpu)` and returning if so. Otherwise, lines 17 and 18 return if this CPU still has not passed through a quiescent state during the current grace period.

Execution reaches line 20 the first time that this CPU is determined to have passed through a quiescent state in the current grace period. Line 20 acquires `rcu_ctrlblk.mutex` to prevent concurrent manipulation of `rcu_ctrlblk.rcu_cpu_mask`. Lines 21 and 22 recheck this CPU's bit under the lock, short-circuiting if it has since cleared. Line 24

clears this CPU's bit, and line 25 invalidates this CPU's quiescent-state snapshot for the benefit of line 13's check for a new grace period on the first invocation of this function during the next grace period. Lines 26 and 27 check to see if this CPU's quiescent state signals the end of the current grace period, short-circuiting if not. Line 29 increments the grace-period counter, signalling other CPUs of the end of the current grace period. Line 30 invokes `rcu_start_batch()` in order to start up the next grace period, if needed. Finally, line 33 releases `rcu_ctrlblk.mutex`.

```
 1 static void rcu_check_quiescent_state(void)
 2 {
 3   int cpu = smp_processor_id();
 4
 5   if (!cpu_isset(cpu, rcu_ctrlblk.rcu_cpu_mask))
 6     return;
 7
 8   /*
 9    * Races with local timer interrupt - in the worst case
10    * we may miss one quiescent state of that CPU. That is
11    * tolerable. So no need to disable interrupts.
12    */
13   if (RCU_last_qsctr(cpu) == RCU_QSCTR_INVALID) {
14     RCU_last_qsctr(cpu) = RCU_qsctr(cpu);
15     return;
16   }
17   if (RCU_qsctr(cpu) == RCU_last_qsctr(cpu))
18     return;
19
20   spin_lock(&rcu_ctrlblk.mutex);
21   if (!cpu_isset(cpu, rcu_ctrlblk.rcu_cpu_mask))
22     goto out_unlock;
23
24   cpu_clear(cpu, rcu_ctrlblk.rcu_cpu_mask);
25   RCU_last_qsctr(cpu) = RCU_QSCTR_INVALID;
26   if (!cpus_empty(rcu_ctrlblk.rcu_cpu_mask))
27     goto out_unlock;
28
29   rcu_ctrlblk.curbatch++;
30   rcu_start_batch(rcu_ctrlblk.maxbatch);
31
32 out_unlock:
33   spin_unlock(&rcu_ctrlblk.mutex);
34 }
```

Figure 4.22: *rcu-ltimer* Check for Quiescent State

Discussion

The counter-and-barrier approach works quite well, and in fact this implementation was accepted into the Linux 2.6 kernel. There are a number of variations on this theme, for

example, the barrier computation may be driven by per-CPU kernel daemons as described in Appendix C.2.1 on Page 330, by per-CPU timers as described in Appendix C.2.2 on Page 332, or by a tasklet as described in Appendix C.2.3 on Page 336. This last implementation also sends interrupts to other CPUs in order to induce them to enter the scheduler. As such, it combines inducing and observing quiescent states.

However, the counter-and-barrier approach makes extensive use of expensive synchronization operations in the barrier computation and requires that read-side RCU critical sections suppress preemption in preemptive environments. The following sections look at some alternative implementations that address these issues.

4.4.2 Counter Ring (rcu-sched)

The *rcu-sched* implementation was designed and implemented by Rusty Russell, with a goal of minimizing `call_rcu()` overhead. This implementation therefore avoids use of any atomic instructions, except as needed to keep a global count of the number of RCU callbacks that are waiting for a grace period to complete. Future work includes implementing this callback counter without use of atomic instructions. Note that the simple split counter described in Section 2.2.13 on Page 41 is not suitable, as it is necessary to read out this callback counter quite frequently, namely, on each and every context switch.

This implementation combines the quiescent-state observation and the barrier computation by using a ring of per-CPU counters, where each CPU sets its counter to one greater than that of its neighbor on each pass through the quiescent state in the scheduler, but only when RCU callbacks are pending. Thus, when a given CPU sees its neighbor's counter change, it is guaranteed that each CPU has passed through the scheduler (a quiescent state) since the given CPU last incremented its own counter, in other words, that a grace period has elapsed.

This implementation has the interesting property that each CPU will be observing a distinct set of grace periods that overlap those of the other CPUs, as shown in Figure 4.23. Here, the vertical lines indicate time passing for each of four CPUs. The dark circles indicate quiescent states, and the numbers denote the value of the per-CPU counter at

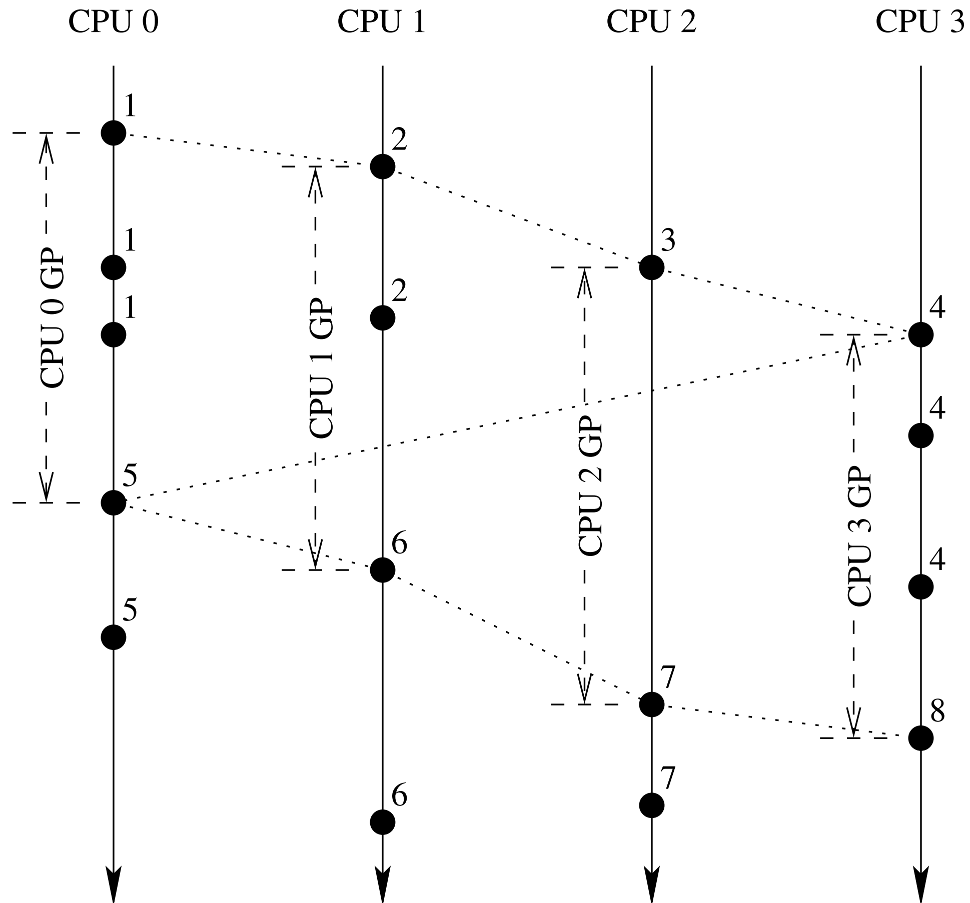


Figure 4.23: Overlapping Grace Periods for Counter Ring

that point in time. The dotted lines connect quiescent states where the per-CPU counter value changed; note that additional quiescent states have no effect in this implementation. Since the definition of a grace period is any time period during which each thread has passed through at least one quiescent state, the time between successive changes in counter value on each CPU is in fact a grace period, but each CPU sees a different grace period. In contrast, in the non-preemptive counter-and-barrier implementations, all CPUs observe identical grace periods.

Note that the ability to insert and remove CPUs, called "CPU hotplug" in Linux, requires an additional per-CPU counter, as shown in Figure 4.24 following the slashes. When CPU 3 is inserted into the system, it must propagate the counter, which will

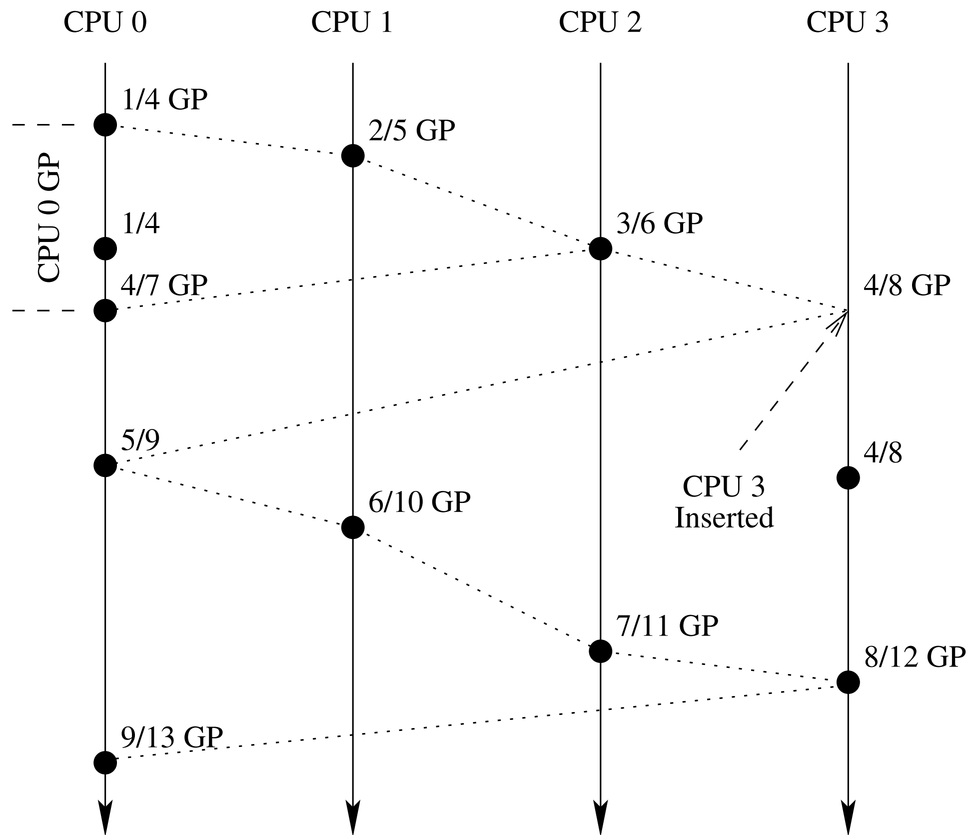


Figure 4.24: Counter Ring and CPU Hotplug

cascade through all the CPUs. The extra counter, named `finished_count`, enables CPUs to determine when a change in counter value does not constitute a grace period. At each context switch, the CPU sets the value of `finished_count` to the current value of the `ring_count` plus the number of active CPUs. On a subsequent context switch, the new value of `ring_count` must exceed that of the old value of `finished_count` for the end of the grace period to be declared.

For example, CPU 0's fourth context switch does *not* mark the end of a grace period, because the new value of `ring_count` (5) does not exceed the old value of `finished_count` (7). This is to be expected, since neither CPU 1 nor CPU 2 have passed through a quiescent state. On the other hand, CPU 1's second context switch *does* mark the end of a grace period, because the new value of `ring_count` exceeds the old value of `finished_count`,

as expected given that all CPUs have context switched since the beginning of the grace period. Note that a grace period must elapse between CPU-hotplug events.

The counter-ring implementation also maintains not just per-CPU callback queues, but two sets of per-CPU-per-IRQ callback queues. This allows the queues to be accessed without the need for either locks (per-CPU) or for interrupt masking (per-IRQ), though this approach has been obsoleted by recent versions of the kernel, which can nest interrupts arbitrarily deeply.[2] One set of these queues accumulates new callbacks from `call_rcu()`, while the other set holds callbacks waiting for the end of the current grace period.

Finally, this implementation places checks in the idle loop in order to ensure that idle CPUs do not indefinitely delay the end of the grace period. This has the beneficial side effect of causing idle-loop execution to be a quiescent state without using the active entities (tasklets, timers, kernel daemons) used by the other implementations.

Figure 4.25 shows the `call_rcu()` function. Lines 9-10 initialize the `rcu_head` callback. Lines 11-14 determine the interrupt state, which is used later as an index to the array of lists of callbacks. Lines 17-18 find the right queue for the callback. The `rcu_batch[cpu].queueing` field is a bit that contains the index of the half of the array that is accumulating new callbacks. The value of this bit is reversed in `rcu_batch_done()` at the end of each grace period. Line 20 increments the number of pending callbacks, which signals the scheduler to start looking for a grace period, and lines 23-24 enqueue the callback.

Figure 4.26 shows the first patch to the scheduler. Lines 12-13 check to see if there are RCU callbacks pending, and, if so, branch to the `rcu_process` label in the second patch shown in Figure 4.27

Lines 8-10 of Figure 4.27 set local variable `c` to one greater than the previous CPU's ring counter. If `c` is different than this CPU's ring count, a grace period has ended, and is handled by lines 16-23. Line 11 checks for scheduler reentry, and if this has not occurred, lines 19-23 invoke `rcu_batch_done()`, protecting against scheduler re-entry by manipulating this CPU's `finished_count`. Line 25 updates this CPU's ring count, which

[2] Therefore, any implementation of this approach in the Linux 2.6 kernel would need to disable interrupts within `call_rcu()`.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *data),
 3               void *data)
 4 {
 5     unsigned cpu = smp_processor_id();
 6     unsigned state;
 7     struct rcu_head **headp;
 8
 9     head->func = func;
10     head->data = data;
11     if (in_interrupt()) {
12         if (in_irq()) state = 2;
13         else state = 1;
14     } else state = 0;
15
16     /* Figure out which queue we're on. */
17     headp = &rcu_batch[cpu].head[
18              rcu_batch[cpu].queueing][state];
19
20     atomic_inc(&rcu_pending);
21     /* Prepend to this CPU's list:
22        no locks needed. */
23     head->next = *headp;
24     *headp = head;
25 }
```

Figure 4.25: *rcu-sched* call_rcu() Implementation

```
 1 @@ -634,10 +639,16 @@
 2      prio_array_t *array;
 3      list_t *queue;
 4      int idx;
 5 +    int c, this_cpu;
 6
 7      if (unlikely(in_interrupt()))
 8          BUG();
 9      release_kernel_lock(prev,
10              smp_processor_id());
11 +
12 +    if (unlikely(is_rcu_pending()))
13 +        goto rcu_process;
14 +
15 +rcu_process_back:
16      spin_lock_irq(&rq->lock);
17
18          switch (prev->state) {
```

Figure 4.26: *rcu-sched* Scheduler Instrumentation, Part 1

will result in the next CPU seeing the end of a grace period. Line 27 returns control to the mainline scheduler.

```
 1 @@ -700,6 +711,23 @@
 2      }
 3      spin_unlock_irq(&rq->lock);
 4
 5 +rcu_process:
 6 +    /* Avoid cache line effects
 7 +       if value hasn't changed */
 8 +    this_cpu = smp_processor_id();
 9 +    c = ring_count((this_cpu + 1) %
10 +           smp_num_cpus) + 1;
11 +    if (c != ring_count(this_cpu)) {
12 +        /* Do subtraction to
13 +           avoid int wrap corner case */
14 +        if (c - finished_count(this_cpu)
15 +                              >= 0) {
16 +            /* Avoid reentry: temporarily
17 +               set finish_count
18 +               far in the future */
19 +            finished_count(this_cpu) =
20 +                           c + INT_MAX;
21 +            rcu_batch_done();
22 +            finished_count(this_cpu) =
23 +                           c + smp_num_cpus;
24 +        }
25 +        ring_count(this_cpu) = c;
26 +    }
27 +    goto rcu_process_back;
28 +
29      reacquire_kernel_lock(current);
30      return;
31  }
```

Figure 4.27: *rcu-sched* Scheduler Instrumentation, Part 2

Figure 4.28 shows how the idle loop is instrumented to prevent architectures that shut down CPUs on idle from indefinitely extending the grace period. The other implementations get this effect through use of timer interrupts or forced context switches, either of which will force the CPU out of the idle loop.

Figure 4.29 shows `rcu_batch_done()`, which is invoked from the scheduler at the end of a grace period. Line 7-8 pick up a pointer to this CPU's set of RCU callback queues. Lines 11-22 invoke all the callbacks in each of this CPU's callback queues (one for each possible IRQ level) that was waiting for the current grace period to expire (selected by `!mybatch->queueing`), and empty each list. Line 25 swaps the sets of queues, so that the callbacks previously waiting for a new grace period to begin are now waiting for the

```
 1 @@ -84,7 +85,8 @@
 2     get into the scheduler unnecessarily.  */
 3     long oldval = xchg(
 4           &current->work.need_resched, -1UL);
 5     if (!oldval)
 6 -       while (current->work.need_resched < 0);
 7 +       while (current->work.need_resched < 0
 8 +              && !is_rcu_pending());
 9     schedule();
10     check_pgt_cache();
11 }
```

Figure 4.28: *rcu-sched* Idle Loop Instrumentation

now-current grace period, and the newly emptied queues will now accept new callbacks registered by future calls to `call_rcu()`.

```
 1 void rcu_batch_done(void)
 2 {
 3     struct rcu_head *i, *next;
 4     struct rcu_batch *mybatch;
 5     unsigned int q;
 6
 7     mybatch =
 8         &rcu_batch[smp_processor_id()];
 9     /* Call callbacks: probably delete
10        themselves, may schedule. */
11     for (q = 0; q < 3; q++) {
12         for (i = mybatch->head[
13                  !mybatch->queueing][q];
14              i;
15              i = next) {
16             next = i->next;
17             i->func(i->data);
18             atomic_dec(&rcu_pending);
19         }
20         mybatch->head[
21            !mybatch->queueing][q] = NULL;
22     }
23
24     /* Start queueing on this batch. */
25     mybatch->queueing = !mybatch->queueing;
26 }
```

Figure 4.29: *rcu-sched* rcu_batch_done()

This implementation has the potential to be extremely efficient, if the need for the global count of RCU callbacks waiting for a grace period can be eliminated. Such elimination, along with modifications required for use in the Linux 2.6 kernel, is future work.

4.4.3 Token Ring with Preemption (K42)

The K42 and Tornado implementations of RCU [30], designed and implemented by researchers at the University of Toronto and IBM Research, are such that read-side critical sections can block as well as being preempted. This means that context switch cannot be used as a quiescent state in these operating-system kernels. Instead, the quiescent states are thread termination and an explicit "quiescent state" operation, similar to a voluntary context switch, used by long-running threads. Unlike Linux, K42/Tornado threads are short lived, with new threads created to process each "message", such as that resulting from a system call, so, unlike Linux, the thread-termination quiescent state is useful within the K42/Tornado kernel. The underlying RCU implementation passes a token among CPUs in a manner similar to the counter ring described in Section 4.4.2. When the token has passed a given CPU twice, a grace period has elapsed.

However, the mapping from threads to CPUs described in Section 3.2.3 on Page 78 does not fully cover this case, since threads may be preempted. This implementation therefore cannot assume that all non-running threads are in a quiescent state. It must therefore explicitly track preempted threads.

This tracking of preempted threads must take into account the fact that preempted threads can migrate among the CPUs. Since there can be many more threads than CPUs, it is again worthwhile to map threads onto CPUs. K42 accomplishes this by having each thread keep track of which CPU it ran on immediately upon being created or regaining control after a voluntary context switch. One can then maintain a per-CPU counter that is atomically incremented each time a thread is created or gains control after a voluntary context switch. Upon termination or beginning the next voluntary context switch, the thread must atomically decrement the same counter that it incremented. Once all counters are zero, no threads are running, indicating the end of a grace period.

However, it is not necessary that *no* thread be running to indicate the end of a grace period, in fact, taking this extreme position would result in infinite grace periods on busy systems. Instead, it is sufficient for all threads that were either running or preempted at the beginning of the grace period to have subsequently executed a voluntary context switch.

K42's implementation of token ring with preemption solves this problem by assigning two counters per CPU. One of the counters is the "current" counter, and it is this counter that each thread atomically increments upon regaining control after creation or a voluntary context switch. Each thread must remember the exact counter it incremented, not just the CPU it started executing on, and each thread is required to atomically decrement this same counter.

When it is necessary to determine when all threads that started on this CPU have completed, the roles of the counters are switched. The counter that was previously being incremented (the "previous" counter) can now only be decremented by threads that had previously incremented it, as threads increment the new "current" counter. Once the "previous" counter reaches zero, all threads that were started on this CPU at the beginning of the grace period have passed through a quiescent state. At this point, the roles of the counters are again switched.

To detect when all CPUs' "previous" counters have reached zero, the CPUs circulate a token. When a CPU receives the token, it forwards the token to the next CPU after two consecutive counter role reversals. Two, rather than one, reversals are necessary because the CPUs see non-overlapping grace periods, similar to the situation described in Section 4.4.2. Therefore, the time between two consecutive possessions of the token by a given CPU is a grace period.

RCU is used pervasively within K42 and Tornado, and is available to user applications and libraries as well as within the kernel. However, system calls such as `recvmsg()`, which can block indefinitely, must use the explicit "quiescent state" operation so that this long-term blocking is not considered to be part of an operation.

Note that K42 and Tornado spawn a new kernel thread to handle each "message", a message being somewhat similar to a Linux system call. The K42/Tornado RCU implementation therefore uses kernel-thread creation and termination as its observed quiescent states, along with a special voluntary-context-switch operation used for long-running message handlers. These quiescent states do not carry over to Linux, which has long-lived kernel threads. Therefore, direct performance and complexity comparisons between the K42/Tornado and the Linux RCU implementations are not meaningful. That said, the

rcu-preempt implementation described in Appendix C.3 on Page 343 was based on the K42/Tornado implementation.

4.5 Single-CPU RCU Implementation

As noted in Chapter 1, single-CPU systems also face synchronization problems. For example, suppose an interrupt handler needs to remove an element from a linked list. If there is a possibility that the mainline code was accessing that element at the time that the interrupt occurred, how does the interrupt handler determine whether it is safe to free that element to the free pool?

One approach is to require that the mainline code disable interrupts around every access to any data structure that might be updated by an interrupt handler. This approach is heavily used in practice. However, disabling and re-enabling interrupts is not free, so it is natural to ask whether it is possible to avoid this cost for read-mostly data structures.

Another approach is for the interrupt handler to defer freeing the removed element. If the element is retained until the mainline code performs a context switch, then there is no danger of the element being freed while it is still in use. Since there is only one CPU, the element could be efficiently freed directly from the context-switch code, if desired. This would be considerably simpler than any of the RCU implementations described in the preceding sections.

However, if an operating system is to run both on SMMP and single-processor systems, software-engineering considerations would motivate use of a single implementation that handled both SMMP and single-processor systems. Both the K42 and Linux systems use a common RCU implementation for both single-processor and multiprocessor operation. It will be interesting to see if either K42 or Linux find it useful to implement a special-case mechanism for single-CPU operation.

4.6 Discussion

This chapter has presented the Linux 2.6 kernel RCU API and a number of implementations of the RCU infrastructure, which are discussed in the following sections.

4.6.1 RCU API Discussion

Although the RCU API has proven quite serviceable in the Linux 2.6 kernel, it is likely to change in the 2.7 effort, if not sooner, for the following reasons:

- Certain heavy networking workloads are able to produce an extremely high rate of deferred destruction in the Linux 2.6.0-test1 kernel when running on a uniprocessor platform. This situation is exacerbated by some issues that permit software-interrupt handlers to run for extremely long periods of time, measuring many seconds in duration. Part of the ongoing fix of course prevents software-interrupt handlers from running so long, but some additional RCU APIs are being considered as well. These APIs are designed to ensure extremely short grace periods for protecting data that are accessed and modified either from interrupt handlers or with interrupts disabled. These primitives might have names like `call_rcu_bh()`, `rcu_read_lock_bh()`, and `rcu_read_unlock_bh()`.[3] These APIs would consider any code sequence with interrupts enabled to be a quiescent state, and would therefore enable faster grace periods for interrupt-oriented algorithms in the newtworking stack and in drivers. More work is needed to determine whether or not these additional APIs are required.

- Some filesystems may have unmount processing that requires all outstanding RCU callbacks for that mount point to complete. A suitable primitive can be easily provided. One API under consideration is `rcu_barrier()`, which waits for all outstanding RCU callbacks to be invoked. One way to implement this is to require that callbacks registered on a given CPU be invoked in the same order that they were registered. There will be interesting interactions with things like CPU hotplug. This is an important area of future work.

- Since the major use of the `call_rcu()` primitive is to defer destruction, one might expect that statements such as `call_rcu(p->head,kfree,p)` would be quite common. This statement frees up the element pointed to by `p` after a subsequent grace period

[3] The "bh" stands for "bottom half", denoting the portion of a driver that does not run in process context. The `_bh` primitives block bottom-half execution.

completes. However, it turns out that directly passing `kfree()` to `call_rcu()` in this manner is quite rare. In most cases, a wrapper function of some sort is needed in order to prepare the data structure being freed, for example, by freeing up structures that it references. Significant memory could be saved by requiring that *all* callbacks be wrapper functions, since such wrapper functions know the location of the `rcu_head` within the structure, eliminating the need for a separate structure pointer to `call_rcu()` and also for the `arg` field of the `rcu_head` structure. This change would simplify the code somewhat, and also reduce RCU's memory overhead.

- When RCU is applied to data structures other than lists, memory-barrier instructions must be explicitly specified. For example, see Mingming Cao's RCU-based implementation of System V IPC, which is described in Section 6.1 on Page 182. Future work includes creating a similar API that encapsulates memory barriers for other data structures.

Evaluation of these changes is ongoing.

4.6.2 RCU Infrastructure Discussion

The RCU infrastructure implementations are summarized in Tables 4.1 and 4.2.

Table 4.1: RCU Implementations Inducing Quiescent States

Name	Batching?	Barrier	Queuing	Quiescent States	Interrupt Handler	Section
simple	N	N/A	global	C	Y	4.3.1
batch	Y	TBD	global	C	Y	4.3.2
rcu-taskq	Y	daemons	global	C	Y	C.1

All these implementations except *rcu-preempt* and *K42* assume a nonpreemptible kernel. Section C.3 on Page 343 describes *rcu-preempt*, which operates efficiently in a preemptive kernel.

Table 4.2: RCU Implementations Observing Quiescent States

Name	Batching?	Sense	Barrier	Queuing	Quiescent States	Interrupt Handler	Section
rcu-ltimer	Y	counters	tasklets	per-CPU	IUC	Y	4.4.1
X-rcu	Y	counters	timers	per-CPU	IC	Y	C.2.2
rcu-krcud	Y	counters	daemons	per-CPU	C	Y	C.2.1
rcu-poll	Y	counters	tasklet	global	C	y	C.2.3
rcu-sched	Y	counter ring	N/A	per-CPU	IC	Y	4.4.2
rcu-preempt	Y	counters	timers	per-CPU	IC	Y	C.3
K42	Y	counter ring	N/A	per-CPU	T	Y	4.4.3

The "Quiescent States" column lists the observed quiescent states that each algorithm tracks, "I" for idle-loop execution, "C" for context switch, "U" for user-mode execution, and "T" for kernel-thread creation and destruction.

The "Interrupt Handler" column indicates whether code in interrupt handlers may safely delete elements of an RCU-protected data structure that is accessed by base-level code with interrupts enabled. The *rcu-poll* implementation may be used in the "bottom half" of Linux drivers, but is interrupt-handler-unsafe by choice, in order to eliminate the overhead of interrupt disabling and enabling that would otherwise be incurred on each call to `call_rcu()`. If a strong need arises for use of `call_rcu()` from interrupt context, trivial changes to *rcu-poll* will render it interrupt-handler-safe.

The RCU implementations discussed in this chapter choose different points in this design space. These implementations are freely available from the Linux Scalability Effort website (http://prdownloads.sf.net/lse/). The *X-rcu*, *rcu-krcud*, and *rcu-ltimer* implementations are similar to the DYNIX/ptx™ implementation, using per-CPU timers, kernel daemons, and architecture-dependent timer support, respectively. The *rcu-taskq* implementation is an extremely compact implementation in which a kernel task forces per-CPU kernel daemons to run on their respective CPUs. The *rcu-sched* implementation uses ring

counters within the Linux scheduler, and boasts an extremely low overhead `call_rcu()` implementation. It is also the only known RCU implementation that uses absolutely *no* locks, interrupt masking, memory barriers, or atomic instructions. The *rcu-poll* implementation is designed for minimal overhead when there are no outstanding RCU callbacks, and boasts very low `call_rcu()` latencies. The *rcu-preempt* implementation adapts the *rcu-krcud* implementation to work correctly in preemptible kernels. Finally, the *K42* implementation uses a dual counter-ring approach [30].

The *rcu-taskq* implementation was the simplest of the non-trivial Linux RCU implementations, *rcu-poll* had the best grace-period latency, and *rcu-ltimer* had the smallest system-wide overhead. Since no current RCU use in Linux requires sub-millisecond grace-period latencies, and since the size of the *rcu-ltimer* diffs are only a few hundred lines larger than the smallest non-trivial patch, the *rcu-ltimer* implementation was accepted into the Linux 2.5.43 kernel [121].

Chapter 5

Design Patterns for RCU

Since RCU is not intended to replace all existing synchronization mechanisms, it is necessary to know when and how to use it. From a performance standpoint, it is clear that RCU is best suited for read-mostly data structures, and the relevant performance tradeoffs are discussed at length in Chapters 7 and 8. However, there are also software-engineering implications surrounding the use of RCU, and it is these implications that are addressed by this chapter via design patterns.

Coplien and Schmidt [21] define "design pattern" as follows; similar definitions may be found elsewhere [3, 28]:

> Design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context. Because they address fundamental challenges in software system development, design patterns are an important technique for improving the quality of software. Key challenges addressed by design patterns include communication of architectural knowledge among developers, accommodating a new design paradigm or architectural style, and avoiding development traps and pitfalls that are usually learned only by (painful) experience.

RCU certainly qualifies as a new design paradigm, so it is reasonable to expect RCU-related design patterns. Such patterns do in fact exist; ten of them are presented in this chapter.

This chapter presents two types of RCU-related patterns. The first type is the RCU design pattern, presented in Section 5.2, which describes situations in which RCU may

easily be applied. The second type is the RCU transformational pattern, presented in Section 5.3, which describes how to transform algorithms not directly amenable to RCU into forms to which RCU may easily be applied. Prior to this, Section 5.1 describes an example algorithm that is used to demonstrate many of the patterns presented in this chapter.

5.1 Example Algorithm

A simple hashed lookup table is used to illustrate the patterns in this chapter. This example hashes a specified key, then searches the indicated list of elements and performs operations on those elements. Both the individual elements and the hash table itself may require mutual exclusion.

The data structure for a uniprocessor implementation is shown in Figure 5.1. The

```
1 struct looktab {
2   struct looktab *next;
3   int     key;
4   int     data;
5 };
```

Figure 5.1: Lookup-Table Element

`next` field links the individual elements together, the `key` field contains the search key, and the `data` field contains the data corresponding to that key. This structure may be embellished as needed for a given synchronization mechanism.

A search for the element with a given key might be implemented as shown in Figure 5.2.

5.2 RCU Design Patterns

This section presents four RCU design patterns, which are part of a larger locking-design pattern language. The patterns in this larger language are as follows:

1. Sequential Program (5.2.2)

2. Code Locking (5.2.2)

```
1 /* Header for list of struct looktab's. */
2
3 struct looktab *looktab_head[LOOKTAB_NHASH] =
4   { NULL };
5 #define LOOKTAB_HASH(key) \
6   ((key) % LOOKTAB_NHASH)
7
8 /*
9  * Return a pointer to the element of the
10  * table with the specified key, or return
11  * NULL if no such element exists.
12  */
13
14 struct looktab *
15 looktab_search(int key)
16 {
17   struct looktab *p;
18
19   p = looktab_head[LOOKTAB_HASH(key)];
20   while (p != NULL) {
21     if (p->key == key) {
22       return (p);
23     }
24     p = p->next;
25   }
26   return (NULL);
27 }
```

Figure 5.2: Lookup-Table Search

3. Data Locking (5.2.2)

4. Data Ownership (5.2.2)

5. Parallel Fastpath (5.2.2)

6. Reader/Writer Locking (5.2.2)

7. RCU, which includes these subpatterns:

   (a) **Pure RCU** (5.2.3)

   (b) **RCU Existence Locks** (5.2.4)

   (c) **Reader-Writer-Lock/RCU Analogy** (5.2.5)

   (d) **RCU Readers With NBS Writers** (5.2.6)

8. Hierarchical Locking ([66])

9. Allocator Caches ([66, 80])

10. Critical-Section Fusing (5.2.2)

11. Critical-Section Partitioning (5.2.2)

The non-RCU patterns are briefly summarized in this section; more details are available elsewhere [66]. This author mined the RCU patterns from earlier work on RCU by myself and others, and these patterns are presented in full detail in this section.

Relationships among these patterns are shown in Figure 5.3 and are described in the following paragraphs.

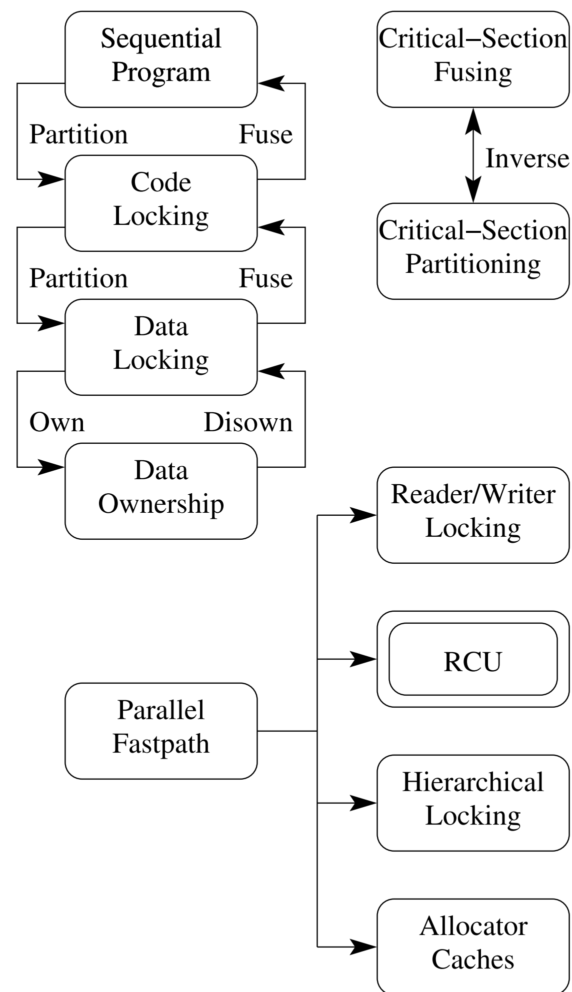


Figure 5.3: Relationship Among Patternssss

Parallel Fastpath and RCU are meta-patterns, that is, they are patterns that describe groups of lower-level patterns. Critical-Section Fusing and Critical-Section Partitioning

are transformational patterns. For example, partitioning a code-locked program's critical sections over instances of an object transforms that program's lock design from code locking to data locking, and vice versa. Similarly, fusing *all* a code-locked program's critical sections results in a sequential program, and vice versa. A data-locking design may be tranformed into a data-ownership design by specifying owning threads (or CPUs) for each data item and coding any needed messaging.

Reader/Writer Locking, RCU, Hierarchical Locking, and Allocator Caches are instances of the Parallel Fastpath meta-pattern. Reader/Writer Locking, RCU, and Hierarchical Locking are themselves meta-patterns; they can be thought of as modifiers to the Code Locking and Data Locking patterns. Parallel Fastpath, Hierarchical Locking, and Allocator Caches are ways of combining other patterns and thus are template patterns.

Critical-Section Partitioning transforms Sequential Program into Code Locking and Code Locking into Data Locking. It also transforms conservative Code Locking and Data Locking into more aggressively parallel forms.

Critical-Section Fusing transforms Data Locking into Code Locking and Code Locking into Sequential Program. It also transforms aggressive Code Locking and Data Locking into more conservative forms.

Assigning a particular CPU or process to each partition of a data-locked data structure results in Data Ownership. A similar assignment of a particular CPU, process, or computer system to each critical section of a code-locked program results in Client/Server, which is used heavily in distributed systems but not discussed further in this paper.

The following section describes the forces that act on locking design patterns, and the ones after that describe the patterns themselves.

5.2.1 Forces

There are a number of forces that act on any programming project, with perhaps the most (in)famous including schedule pressure, elegance, performance, and work-life balance. Very often, perfectly handling one of the forces acting on the project will sacrifice the others, so that the most aggressive possible schedule will likely involve sacrifices in the areas of elegance, performance, and work-life balance.

A given design pattern may be thought of as a particular way of balancing the forces acting on the design, with particularly good patterns describing particularly good balances among those forces [3].

The forces acting on the performance of parallel programs are speedup, contention, overhead, read-to-write ratio, and complexity:

Speedup: Getting a program to run faster is the only reason to go to all of the time and trouble required to parallelize it. Speedup is defined to be the ratio of the time required to run a sequential version of the program to the time required to run a parallel version.

Contention: If more CPUs are applied to a parallel program than can be kept busy by that program, the excess CPUs are prevented from doing useful work by contention.

Overhead: A uniprocessor, single-threaded, non-preemptible, and non-interruptible[1] version of a given parallel program would not need synchronization primitives. Therefore, any time consumed by these primitives (including communication cache misses as well as locking primitives, atomic instructions, and memory barriers) is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the overhead of the code in the critical section, with larger critical sections able to tolerate greater synchronization overhead.

Read-to-Write Ratio: A data structure that is rarely updated may often be protected with asymmetric synchronization primitives that reduce readers' synchronization overhead at the expense of that of writers, thereby reducing overall synchronization overhead. Corresponding optimizations are possible for frequently updated data structures, as discussed in Section 2.2.13 on Page 41.

Complexity: A parallel program is more complex than an equivalent sequential program because the parallel program has a much larger state space than does the sequential

[1]Either by masking interrupts or by being oblivious to them.

program. A parallel programmer must consider synchronization primitives, locking design, critical-section identification, and deadlock in the context of this larger state space.

This greater complexity often translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program, since a given degree of speedup is worth only so much time and trouble.

These forces will act together to enforce a maximum speedup. The first three forces are deeply interrelated, so the remainder of this section analyzes these interrelationships.[2]

Note that these forces may also appear as part of the context. For example, speedup may act as a force ("faster is better") or as part of the context ("the system must achieve a throughput of at least 1,000 transactions per second").

An understanding of the relationships between these forces can be very helpful when resolving the forces acting on an existing parallel program.

1. The less time a program spends in critical sections, the greater the potential speedup.

2. The fraction of time that the program spends in a given critical section must be much less than the reciprocal of the number of CPUs for the actual speedup to approach the number of CPUs. For example, a program running on 10 CPUs must spend much less than one tenth of its time in the critical section if it is to scale well.

3. Contention effects will consume the excess CPU and/or wallclock time should the actual speedup be less than the number of available CPUs. The larger the gap between the number of CPUs and the actual speedup, the less efficiently the CPUs will be used. Similarly, the greater the desired efficiency, the smaller the achievable speedup.

4. If the available synchronization primitives have high overhead compared to the critical sections that they guard, the best way to improve speedup is to reduce the

[2] A real-world parallel system will have many additional forces acting on it, such as data-structure layout, memory size, memory-hierarchy latencies, and bandwidth limitations.

number of times that the primitives are invoked (perhaps by fusing critical sections, using data ownership, using RCU, or by moving toward a more coarse-grained parallelism such as code locking).

5. If the critical sections have high overhead compared to the primitives guarding them, the best way to improve speedup is to increase parallelism by moving to reader/writer locking, data locking, RCU, or data ownership.

6. If the critical sections have high overhead compared to the primitives guarding them and the data structure being guarded is read much more often than modified, the best way to increase parallelism is to move to reader/writer locking or RCU.

5.2.2 Non-RCU Locking Design Patterns

The following sections briefly describe the non-RCU locking design patterns. Readers wishing more detail may consult this previously published locking design pattern language [66].

Sequential Program

If the program runs fast enough on a single processor, and has no interactions with other processes, threads, or interrupt handlers, you should remove the synchronization primitives and spare yourself their overhead and complexity. Some would argue that Moore's Law will eventually force all programs into this category. Others would disagree.

Code Locking

Code locking is the simplest locking design, using only global locks. It is especially easy to retrofit an existing program to use code locking in order to run it on a multiprocessor. If the program has only a single shared resource, code locking will even give optimal performance. However, most programs of any size and complexity require much of the execution to occur in critical sections, which in turn sharply limits the scaling.

Therefore, use code locking on programs that spend only a small fraction of their run time in critical sections or from which only modest scaling is required. In these cases,

code locking will provide a relatively simple program that is very similar to its sequential counterpart.

Data Locking

Many algorithms and data structures may be partitioned into independent parts, with each part of the data structure having its own lock. Then the critical sections for each part of the data structure can execute in parallel, although only one instance of the critical section for a given part could be executing at a given time. Use data locking when contention must be reduced, and where synchronization overhead is not limiting speedups. Data locking reduces this overhead by distributing the instances of the overly-large critical section into multiple critical sections.

Data Ownership

Data ownership partitions a given data structure over the threads or CPUs, so that each thread/CPU accesses its subset of the data structure without any synchronization overhead whatsoever. However, if one thread wishes to access some other thread's data, the first thread is unable to do so directly. Instead, the first thread must communicate with the second thread, so that the second thread performs the operation on behalf of the first, or, alternatively, migrates the data to the first thread.

Data ownership might seem arcane, but it is used very frequently:

1. Any variables accessible by only one CPU or thread (such as `auto` variables in C and C++) are owned by that CPU or process.

2. An instance of a user interface owns the corresponding user's context. It is very common for applications interacting with parallel database engines to be written as if they were entirely sequential programs. Such applications own the user interface and his current action. Explicit parallelism is thus confined to the database engine itself.

3. Parametric simulations are often trivially parallelized by granting each thread ownership of a particular region of the parameter space.

If there is significant sharing, communication between the threads or CPUs can result in significant complexity and overhead. However, in situations where no sharing is required, data ownership achieves ideal performance. Such situations are commonly referred to as "embarassingly parallel".

Parallel Fastpath

The idea behind the Parallel Fastpath design pattern is to aggressively parallelize the common-case code path without incurring the complexity that would be required to aggressively parallelize the entire algorithm. You must understand not only the specific algorithm you wish to parallelize, but also the workload that the algorithm will be subjected to. Great creativity and design effort is often required to construct a parallel fastpath.

Parallel fastpath combines different patterns (one for the fastpath, one elsewhere) and is therefore a template pattern. The following instances of parallel fastpath occur often enough to warrant their own patterns:

1. Reader/Writer Locking (5.2.2).
2. Pure RCU (5.2.3).
3. Reader-Writer-Lock/RCU Analogy (5.2.5).
4. RCU Existence Locks(5.2.4).
5. RCU Readers With NBS Writers (5.2.6).
6. Hierarchical Locking ([66]).
7. Resource Allocator Caches ([66, 80]).

Reader/Writer Locking

If synchronization overhead is negligible (i.e., the program uses coarse-grained parallelism), and only a small fraction of the critical sections modify data, then allowing multiple readers

to proceed in parallel can greatly increase speedup. Writers exclude both readers and each other.

Reader/writer locking is a simple instance of asymmetric locking. Snaman [112] describes a more ornate six-mode asymmetric locking design used in several clustered systems. Asymmetric locking primitives can be used to implement a very simple form of the **Observer Pattern** [28]–when a writer releases the lock, all readers are notified of the change in state.

Critical-Section Fusing

If the overhead of the code between two critical sections is less than the overhead of the synchronization primitives, fusing the two critical sections will decrease overhead and increase speedups.

Critical-section fusing is a meta-pattern that transforms Data Locking into Code Locking and Code Locking into Sequential Program. In addition, it transforms more-aggressive variants of Code Locking and Data Locking into less-aggressive variants.

Critical-section fusing is the inverse of Critical-Section Partitioning.

Critical-Section Partitioning

If the overhead of the non-critical-section code inside a single critical section is greater than the overhead of the synchronization primitives, splitting the critical section can decrease overhead and increase speedups.

Critical-section partitioning is a meta-pattern that transforms Sequential Program into Code Locking and Code Locking into Data Locking. It also transforms less-aggressive variants of Code Locking and Data Locking into more-aggressive variants.

Critical-section partitioning is the inverse of Critical-Section Fusing.

5.2.3 Pure RCU

Pure RCU describes how to apply RCU to speed up read-only accesses in cases where stale and inconsistent data may be tolerated. This pattern is especially useful in cases where all outstanding interrupt handlers must complete before an update may be finalized.

Problem: How can programs that do not require consistency and freshness guarantees improve their performance and/or reduce their complexity?

Context: An existing program that uses locking or non-blocking synchronization, but has the following properties:

1. Runs in an environment consisting of short, quickly completed units of work, for example, an operating system kernel, a server application, or an event-driven real-time system.
2. Can tolerate stale data, perhaps by rejecting it as part of higher-level processing.
3. Can tolerate inconsistent data, perhaps requiring that readers reference a given data item only once and that writers make atomic updates.

These properties can be tolerated in a surprisingly large number of situations, and many less-tolerant algorithms may be easily transformed to permit Pure RCU to be used, as described in Section 5.3. Examples of situations that tolerate stale and inconsistent data include routing tables, data structures maintaining hardware and software configuration information, decision-support heuristics, and mathematical algorithms with convergence checks.

Forces:

- Read-to-Write Ratio (+++): Pure RCU not only permits read-side parallelism, but in addition incurs zero read-side synchronization overhead. This permits Pure RCU to offer excellent performance in read-mostly situations.
- Speedup (+++): Since readers proceed in parallel, very high speedups are possible.
- Contention (++): Since readers contend neither with each other nor with writers, contention is low.
- Overhead (++): Read-side code requires no synchronization primitives of any kind, though read-side memory barriers *are* required on DEC Alpha, due to its extremely weak memory-consistency model, as described in Appendix B on Page 322.

- Complexity $(-?)$: The Pure RCU approach can add some complexity in many cases, but actually simplifies the following aspects due to the fact that readers need acquire no locks:

  1. Locking hierarchies.
  2. Handling of deadlock issues. In particular, non-maskable interrupts may use Pure RCU in a very natural manner.
  3. Existence locks, which may often be eliminated entirely.

Solution: Use Pure RCU to improve speedups and, in some cases, reduce complexity of programs that rarely modify shared data. Pure RCU is not sufficient for the hashed lookup table described in Section 5.1 due to the need to provide memory barriers, instead, apply the Reader-Writer-Lock/RCU Analogy pattern from Section 5.2.5.

Pure RCU can be best illustrated by an interrupt-shutdown problem. Suppose a networking device is in operation, and that there are a number of dynamically allocated data structures used by that device's interrupt handlers. The device's interrupts are turned off as part of the shutdown procedure, but on a multiprocessor, the handlers for earlier interrupts from that device may still be executing. How can we know when it is safe to free up the interrupt handler's data structures without requiring each interrupt to undertake an expensive lock acquisition?[3]

The Pure RCU solution to the interrupt race problem is to:

1. Turn off the device's interrupts.
2. If necessary, wait for any interrupts propagating through the device to reach the CPU (this is often accomplished either by a read from the device's registers or by a device reset).
3. Use `synchronize_kernel()` to block for one grace period. At the end of the grace period, all earlier interrupts must have completed.

[3]Since devices are shut down infrequently, the shutdown procedure can incur substantial overhead. In contrast, interrupts occur quite frequently, and therefore tolerate only minimal synchronization overhead, preferably none at all.

4. Thus, it is now safe to tear down and free up data structures accessed by that device's interrupt handlers.

This same approach may be used to handle rare changes in mode of operation [111], for example, changing to recovery mode in a cluster after node failure.

Without RCU, complex and inefficient locking schemes must be imposed on interrupt handlers in order to be able to reliably shut down interrupts and free up the relevant data structures [30].

Resulting Context: A program that allows interrupt handlers to proceed safely, without needing to perform high-overhead synchronization operations. Writers will run concurrently with readers, so that readers may see inconsistent intermediate states. In situations where this is a problem, it may be resolved using one of the design patterns described in Section 5.3 on Page 159.

Design Rationale: If a program is rarely modifying a data structure, why should the readers be required to use expensive synchronization operations to handle the rare case of updates?

Example Uses: Pure RCU is used by

1. DYNIX/ptx's LAN driver to mediate races between LAN device shutdown and packet-reception interrupts from that device [73].
2. The Linux 2.6 kernel's IPMI and NMI handlers, the latter of which is described in detail in Section 6.3 on Page 211.
3. Linux's module-unloading code, both in the 2.6 kernel and in the 2.4-based SuSE 7.3 Update and United Linux kernels, as described in Section 6.4 on Page 213.
4. K42's hot-swapping infrastructure [10].

5.2.4 RCU Existence Locks

RCU Existence Locks defer freeing of data-structure elements so that readers may traverse pointers from one element to the next without holding explicit "existence locks" that would otherwise be required to ensure that the target element was not prematurely freed. RCU Existence Locks can greatly simplify locking designs, since explicit existence locks can be complex and prone to deadlock situations [30].

Problem: How can complex existence-locking or reference-counting designs be simplified?

Context: An existing program that uses locking or reference counts to ensure that data structures are not prematurely recycled.

Forces:

- Read-to-Write Ratio (+ + +): Expensive read-side locking and reference counting is eliminated.
- Overhead (++): The overhead of read-side locks and reference counts is reduced.
- Speedup (+): Elimination of read-side locking and reference counting can eliminate some scaling bottlenecks.
- Contention (+): Contention of read-side locks is reduced, either by acquiring them less often or by no longer performing expensive reference counting in their critical sections.
- Complexity (+): The severity of deadlock issues is reduced by having fewer lock acquisitions, and complex reference-count interactions are eliminated.

Solution: Suppose that a deletion primitive is needed for the hash-table solution presented in the Reader-Writer-Lock/RCU Analogy pattern in Section 5.2.5 on Page 153. The deleted entry must not be freed up until all concurrent readers have finished with that

entry. The usual way to accomplish this is to require the readers to use explicit existence locks or reference counts, both of which impose expensive synchronization operations on the readers.

An alternative is to use the RCU `synchronize_kernel()` primitive as shown in Figure 5.4, which prevents the entry from being freed until all racing readers have completed their search.

```
 1 /*
 2  * Delete a looktab element.
 3  * Return TRUE if successful,
 4  * FALSE if not found
 5  */
 6
 7 int
 8 looktab_delete(int key)
 9 {
10   struct looktab **p;
11   struct looktab *q;
12
13   spin_lock(&looktab_mutex);
14   p = &looktab_head[LOOKTAB_HASH(key)];
15   while (*p != NULL) {
16     if ((*p)->key == key) {
17       q = *p;
18       *p = (*p)->next;
19       spin_unlock(&looktab_mutex);
20       synchronize_kernel();
21       kfree(q);
22       return (TRUE);
23     }
24     p = &(*p)->next;
25   }
26   spin_unlock(&looktab_mutex);
27   return (NULL);
28 }
```

Figure 5.4: RCU Existence Locks

Resulting Context: A program where readers may proceed without synchronization primitives, but elements may be concurrently deleted safely, without need for explicit existence locks or reference counts. Readers may see stale data, and the transformational design patterns presented in Section 5.3 may be used to address this if required.

Design Rationale: Deferring destruction of data structures until all readers are known to be done with them eliminates any need for complex existence locking or reference

counting schemes. As the K42 and Tornado experiences have demonstrated, eliminating existence locks in turn greatly reduces or even eliminates the need for complex locking hierarchies and deadlock avoidance [30], thereby greatly simplifying the operating system's implementation.

Example Uses: RCU Existence Locks is used as follows:

1. The Linux 2.6 kernel's System V IPC implementation, described in Section 6.1 on Page 182.
2. The Linux 2.6 kernel's directory-entry-cache implementation, as described in Section 6.2 on Page 195.
3. The Linux 2.6 kernel's IP route cache, described by McKenney et al. [78].
4. The Linux tasklist-lock patch, described in Section 6.5 on Page 213.
5. The Linux FD management patch, described in Section 6.6 on Page 218.
6. K42's non-blocking hash tables, as described in Section 6.7 on Page 221.
7. K42's hot-swapping infrastructure [10]. In addition RCU Existence Locks is used pervasively throughout the K42 research operating system.

5.2.5 Reader-Writer-Lock/RCU Analogy

Reader-Writer-Lock/RCU Analogy describes how to convert an existing reader-writer-lock-based algorithm to use RCU, but only in cases where stale and inconsistent data may be tolerated.

Problem: How can programs that are experiencing excessive read-side contention and cache thrashing improve their performance?

Context: An existing program that uses reader-writer locking, and has the following properties:

1. Runs in an environment consisting of short, quickly completed units of work.
2. Can tolerate stale data, perhaps by rejecting it as part of higher-level processing.
3. Can tolerate inconsistent data, perhaps by doing only atomic updates.

Forces:

- Read-to-Write Ratio (+ + +): Reader-Writer-Lock/RCU Analogy not only permits read-side parallelism, but in addition incurs zero read-side synchronization overhead. This permits Reader-Writer-Lock/RCU Analogy to offer excellent performance in read-mostly situations.
- Speedup (+++): Since readers proceed in parallel, very high speedups are possible.
- Contention (++): Since readers contend neither with each other nor with writers, read-side contention is low.
- Overhead (++): Read-side code requires no synchronization primitives of any kind (though read-side memory barriers *are* required on DEC Alpha, due to its extremely weak memory-consistency model).
- Complexity (0): As is shown below, this pattern does a one-for-one substitution of RCU primitives for reader-writer locking primitives, so the structure of the code remains the same. However, since readers no longer acquire locks, the potential for deadlock decreases. This is balanced by the need to consider races between readers and writers.

Solution: Although RCU has been used in a great many interesting and surprising ways, one of the most straightforward is as a replacement for reader-writer locking. This section demonstrates this replacement in the Linux 2.6 kernel. Figure 5.5 shows locking for search, and Figure 5.6 shows locking for update.

Note that searches can race with updates, so the update must be carried out in such a manner that all intermediate states are safe to search. If necessary, use the transformation design patterns in Section 5.3 in order to make update algorithms meet this requirement.

```
 1 /*
 2  * Look up a looktab element and
 3  * examine it.
 4  */
 5
 6 rcu_read_lock();
 7 p = looktab_search(mykey);
 8
 9 /*
10  * insert code here to examine
11  * the element.
12  */
13
14 rcu_read_unlock();
```

Figure 5.5: RCU Read-Side Code

```
 1 /*
 2  * Global lock for struct looktab
 3  * manipulations.
 4  */
 5
 6 spinlock_t looktab_mutex;
 7
 8 . . .
 9
10 /*
11  * Look up a looktab element and
12  * examine it.
13  */
14
15 spin_lock(&looktab_mutex);
16 p = looktab_search(mykey);
17
18 /*
19  * insert code here to update
20  * the element.
21  */
22
23 spin_unlock(&looktab_mutex);
```

Figure 5.6: RCU Write-Side Locking

The Reader-Writer-Lock/RCU Analogy substitutes primitives as shown in Table 5.1. The asterisked primitives are no-ops in non-preemptible kernels; in preemptible kernels, they suppress preemption, which is normally an extremely cheap operation on the local task structure. Note that since neither `rcu_read_lock()` nor `rcu_read_unlock` block irq or softirq contexts, it is necessary to add primitives for this purpose where needed. For example, `read_lock_irqsave` must become `rcu_read_lock()` followed by `local_irq_save()`. The last entry for `kfree()` is strictly speaking from the RCU Existence Locks pattern, but is almost always combined with the Reader-Writer-Lock/RCU Analogy.

Table 5.1: Reader-Writer-Lock/RCU Substitutions

Reader-Writer Lock	RCU
`rwlock_t`	`spinlock_t`
`read_lock()`	`rcu_read_lock()` *
`read_unlock()`	`rcu_read_unlock()` *
`write_lock()`	`spin_lock()`
`write_unlock()`	`spin_unlock()`
`list_add()`	`list_add_rcu()`
`list_add_tail()`	`list_add_tail_rcu()`
`list_del()`	`list_del_rcu()`
`list_for_each()`	`list_for_each_rcu()`
`kfree()`	`call_rcu(kfree)`

* no-op unless CONFIG_PREEMPT, in which case preemption is suppressed

Although this pattern can be quite compelling and useful, there are some caveats:

1. Read-side critical sections may see "stale data," that has been removed from the list but not yet freed. There are some situations (e.g., routing tables for best-effort protocols) where this is not a problem. In other situations, the transformational design patterns described in Section 5.3 may be used to render the algorithm tolerant of stale data.

2. Read-side critical sections may run concurrently with write-side critical sections, and thus see inconsistent data due to an in-progress update. In situations where this is a problem, the transformational design patterns described in Section 5.3 may be applied to the algorithm to render it tolerant of stale data.

3. The grace period will delay freeing of memory, which means that both the memory and the cache footprint of the code will be somewhat larger when using RCU than when using reader-writer locking.

Note that this pattern may be applied incrementally, so that critical read-side code uses RCU, but the remainder of the read-side code still acquires the reader-writer lock. For example, Section 6.5 on Page 213 presents incremental application of this pattern to the Linux tasklist structure.

Where it applies, this transformation pattern can deliver full parallelism with almost no increase in complexity. For example, Section 6.1 on Page 182 shows how applying this transformation pattern to System V IPC yields order-of-magnitude speedups with a very small increase in code size and complexity.

Resulting Context: A program very similar to its reader-writer-locked predecessor, but with read-side code free of expensive synchronization operations.

Design Rationale: Do not incur the expense of excluding writers when it is not necessary.

Example Uses: Reader-Writer-Lock/RCU Analogy is used heavily in DYNIX/ptx. It is also used in the Linux 2.6 kernel:

1. The System V IPC implementation, described in Section 6.1 on Page 182.
2. The directory-entry cache (dcache), described in Section 6.2 on Page 195.
3. The IP route cache, described by McKenney et al. [71].
4. The tasklist patch described in Section 6.5 on Page 213.
5. The FD management patch described in Section 6.6 on Page 218.

In most of these cases, Reader-Writer-Lock/RCU Analogy is used in combination with RCU transformational design patterns. For example, the `_rcu` variants of the list macros use Ordered Update With Ordered Read by adding internal memory barriers as needed on

particular CPU architectures. As noted earlier, many uses of Reader-Writer-Lock/RCU Analogy change `kfree()` calls to use `call_rcu()` in order to defer the free, in accordance with the RCU Existence Locks design pattern.

5.2.6 RCU Readers With NBS Writers

RCU Readers With NBS Writers uses non-blocking synchronization rather than locking for updates. However, use of RCU simplifies the update code by guaranteeing that deleted elements will not be freed while readers hold references to them.

Problem: How can programs with complex non-blocking synchronization (NBS) be simplified?

Context: An existing program that uses NBS on read-mostly data structures.

Forces:

- Read-to-Write Ratio (+ + +): Expensive read-side synchronization operations are eliminated.
- Overhead (++): Eliminating read-side synchronization operations also eliminates their overhead.
- Complexity (++): Since the underlying NBS algorithm no longer needs to explicitly maintain existence criteria, the algorithm is greatly simplified.
- Speedup (+): Eliminating read-side synchronization operations can eliminate some scaling bottlenecks.
- Contention (+): Contention of underlying LL/SC or compare-and-swap operations against the cache lines that they operate on is reduced.

Solution: The read-side NBS code is replaced by code that traverses the relevant data structures as if they were statically allocated. NBS code can be quite complex, so the full solution for hash tables is described in Section 6.7 on Page 221.

Resulting Context: A program where readers may proceed without synchronization operations, but with writers using NBS algorithms. Readers may see stale data, but the transformational design patterns presented in Section 5.3 may be used to address this if required.

Design Rationale: Deferring destruction of data structures until all readers are known to be done with them eliminates any need for complex existence-maintenance schemes.

Example Uses: RCU Readers With NBS Writers is used by K42's non-blocking hash-table implementation, which is describe in Section 6.7 on Page 221. Since there is only one use, this is a provisional pattern.

5.3 Patterns for Transforming Algorithms to RCU

The basic RCU infrastructure has comparatively limited applicability. This chapter presents patterns that greatly extend RCU's reach by transforming algorithms to tolerate RCU's stale-data and inconsistency properties. These tranformation patterns enable RCU to be used on a wide variety of real-world problems, as will be shown in Chapter 6. This author mined the following transformational patterns from earlier uses of RCU by myself and others:

1. Mark Obsolete Objects,
2. Substitute Copy For Original,
3. Impose Level Of Indirection,
4. Ordered Update With Ordered Read,
5. Global Version Number, and
6. Stall Updates.

Each of these patterns is discussed in a later section.

This wide variety of transformational patterns will usually require ad hoc examples, but, where applicable, the example described in Section 5.1 will be used.

5.3.1 Forces

Although the forces described in Section 5.2.1 still apply, the more specific set of forces described below is better suited for this lower-level pattern language.

The purpose of this pattern language is to transform algorithms into a form more amenable to use with RCU, by making them tolerant of stale and inconsistent data, or by making them perform atomic updates to prevent inconsistent data from ever appearing. In some cases, these transformations come at a cost, in the form of (preferably lightweight) read-side synchronization or mutual-exclusion techniques or read-side memory barriers.

The forces, which in this case can also be thought of as attributes, are thus as follows:

Fresh: Since RCU readers do not exclude writers, readers can find themselves referencing old versions of the data, or "stale data". A pattern that addresses this force therefore transforms the algorithm into a form that rejects stale data, so that only fresh data is actually used. For example, in Mark Obsolete Object, readers reject any object that has been marked as obsolete.

Consistent: Again, RCU readers do not exclude writers, so that readers may see inconsistencies if writers make non-atomic changes or if readers access the same data multiple times. A pattern that addresses this force transforms the algorithm into a form that sees only consistent data, either by making changes atomically (for example, Substitute Copy For Original) or by rejecting inconsistencies (for example, Global Version Number).

Atomic: Atomicity prevents some types of inconsistency. A pattern that addresses this force transforms the algorithm from a form in which readers can see partially completed updates into a form where updates appear atomic to readers.

Mutex: The presence of this force indicates that readers must perform some sort of mutual exclusion, involving something like locking or atomic instructions. This mutual exclusion will be reasonably light weight, for example, Mark Obsolete Objects requires per-element mutual exclusion but no global locks. Nonetheless, the added

read-side overhead will cause the resulting program to more poorly resolve the upper-level Read-to-Write Ratio, Speedup, and Overhead forces.

Memory Barrier: The presence of this force indicates that readers must execute a memory-barrier instruction on CPUs with weak memory consistency models. For example, in the Global Version Number pattern, the readers must use memory barriers to ensure that the first snapshot of the version number is taken before any other accesses, and that the last snapshot is taken after any other accesses. Again, the added read-side overhead will cause the resulting program to more poorly resolve the upper-level Read-to-Write Ratio, Speedup, and Overhead forces. In addition, the memory barriers result in additional Complexity.

The Fresh and Consistent forces permit operation with RCU, while the Atomic force permits additional transformational patterns, such as Global Version Number, to be applied.

5.3.2 Mark Obsolete Objects

Mark Obsolete Objects transforms an algorithm that cannot tolerate stale data into one that is able to do so by marking deleted elements. Readers can then ignore any elements that are so marked.

Problem: How can an algorithm that cannot tolerate stale data be made to work with RCU?

Context: An algorithm that uses a coarsely locked search structure mapping to elements that can be protected via Data Locking, where it is desireable to eliminate contention or cache thrashing on the lock protecting the search structure. The search structure is thus a read-mostly mapping to often-written data elements.

Forces:

- **Fresh**, since data elements marked "obsolete" will be ignored.

- Not **Consistent** in general. Consistent if data elements are accessed only singly. If consistency across multiple data elements is required, use the Substitute Copy For Original on the whole group of data elements, or use Stall Updates.
- Not **Atomic** though atomic update is often provided by the Data Locking pattern.
- Uses **Mutex** primitives through the Data Locking pattern, in order to reliably sample the "obsolete" marking.
- Uses **Memory Barrier** primitives to reliably sample the "obsolete" marking.

Solution: Use RCU to replace the lock on the search structure. If a reader finds an element marked obsolete, it must act as if the search failed, as shown in Figure 5.7 and 5.8. The fact that the reader holds the element's mutex ensures that it will not be marked obsolete while it is being accessed.

```
 1 /* Global lock for struct looktab updates. */
 2
 3 spinlock_t my_looktab_mutex;
 4 struct looktab *my_looktab[LOOKTAB_NHASH];
 5
 6 . . .
 7
 8 /* Look up a looktab element and examine it. */
 9
10 p = looktab_search(my_looktab, mykey);
11 if (p == NULL) {
12   /* insert code here to handle search failure. */
13 } else {
14   /* insert code here to examine or update the element. */
15 }
16 spin_unlock(&p->mutex);
```

Figure 5.7: Mark Obsolete Object Reader

However, this read-side access algorithm requires that the `looktab_search()` function acquire the element's mutex, as shown in Figure 5.8. This increased overhead is usually more than made up for by the removal of the lock guarding the search structure.

Elements must be marked obsolete upon deletion, as shown in Figure 5.9. Note that the `synchronize_kernel()` invocation is required in order to ensure that the deleted

```
 1 struct looktab *
 2 looktab_search(int key)
 3 {
 4   struct looktab *p;
 5
 6   p = looktab_head[LOOKTAB_HASH(key)];
 7   while (p != NULL) {
 8     if ((p->key == key) &&
 9         !p->obsolete) {
10       spin_lock(&p->mutex);
11       return (p);
12     }
13     p = p->next;
14   }
15   return (NULL);
16 }
```

Figure 5.8: Mark Obsolete Object Search

element remains in place until all concurrent readers drop any references to it. Note also the use of the global lock to serialize concurrent deletions.

Resulting Context: A program that rejects stale data, thus allowing use of RCU for the search structure.

Design Rationale: Explicitly marking obsolete data removes the need to lock the search structure, thereby improving scalability and performance.

Example Uses: The Mark Obsolete Objects pattern is used in the:

1. Linux 2.6 kernel's System V IPC implementation, as described in Section 6.1 on Page 182.

2. K42 hash table implementation, as described in Section 6.7 on Page 221.

It has also seen much use in non-open environments.

5.3.3 Substitute Copy For Original

Substitute Copy For Original transforms an algorithm that cannot tolerate inconsistent data into one that can, by hiding non-atomic updates behind an atomic substitution operation.

```
 1 /*
 2  * Delete a looktab element.
 3  * Return TRUE if successful,
 4  * FALSE if not found
 5  */
 6
 7 int
 8 looktab_delete(int key)
 9 {
10   struct looktab **p;
11   struct looktab *q;
12
13   spin_lock(&looktab_mutex);
14   p = &looktab_head[LOOKTAB_HASH(key)];
15   while (*p != NULL) {
16     if ((*p)->key == key) {
17       q = *p;
18       *p = (*p)->next;
19       spin_lock(&p->mutex);
20       q->obsolete = TRUE;
21       spin_unlock(&p->mutex);
22       spin_unlock(&looktab_mutex);
23       synchronize_kernel();
24       kfree(q);
25       return (TRUE);
26     }
27     p = &(*p)->next;
28   }
29   spin_unlock(&looktab_mutex);
30   return (NULL);
31 }
```

Figure 5.9: Mark Object Obsolete Upon Deletion

Problem: How do you make complex updates appear atomic to readers?

Context: A program that requires complex updates, but also needs lock-free searches in order to improve performance.

Forces:

- Not **Fresh**, since readers might still be accessing the original after the copy has been substituted.
- **Consistent**, since the copy is initialized to be consistent before being substituted, but only as long as readers dereference the pointer to the data at most once per read-side critical section.
- **Atomic**, since writes of aligned pointers are atomic.
- Does not use **Mutex** since writes of aligned pointers are atomic.
- Does not use **Memory Barrier**, except on DEC Alpha, since there is a data dependency between accessing the pointer and dereferencing it. See Appendix B on Page 322 for more details.

Solution: The solution is to allocate a new element, copy the old element to it, perform the updates on the new element, execute any needed memory barrier, and substitute the new element for the old one, as shown in Figure 5.10. Finally, after a grace period, the old element is freed up. Readers will see either the old element, or the new one, but nothing in between. Readers need neither mutual exclusion nor memory barriers, except on DEC Alpha, where memory barriers are required as discussed in Appendix B on Page 322.

Figure 5.11 shows the corresponding read-side code. The do-while loop retries the access should stale or inconsistent data be detected. The body of the loop first takes a snapshot of the `foop` pointer, then executes an Alpha-only memory barrier. On other CPUs, the data dependency dereferencing the `foop` pointer acts as an implicit memory barrier, so that `smp_read_barrier_depends()` is a no-op on these other CPUs. The code then does whatever accesses are required, followed by another memory barrier, which

```
 1 /* Global pointer that serves as commit point.  */
 2
 3 struct foo *foop;
 4 . . .
 5
 6 /* First allocate and copy from old. */
 7
 8 p = kmalloc(sizeof(*p));
 9 *p = *foop;
10
11 /* Modify any required fields to create new. */
12
13 p->field1 = value1;
14 p->field2 = value2;
15
16 /* Update: issue memory barrier. */
17
18 smp_wmb();
19
20 /* Update: update pointer. */
21
22 spin_lock(&looktab_mutex);
23 q = foop;
24 foop = p;
25 spin_unlock(&looktab_mutex);
26 synchronize_kernel();
27 kfree(q);
```

Figure 5.10: Substituting a Copy For Original

ensures that the do-while's condition will not be evaluated until after the the accesses are complete. The do-while condition forces re-execution if the pointer changed during the access.

This technique may be applied to larger multilinked structures, though the cost of the updates increases with size. If there are many possible pointers to the structure being replaced, then all the pointers must be updated to point to the new copy. If a reader can traverse multiple pointers into the structure being substituted, then that reader might see different versions on different traversals. This pattern is therefore most applicable to algorithms that perform acyclic traversals of linked data structures.

Resulting Context: A program where updates are atomic, so that RCU may be used safely.

Design Rationale: The key concept is to hide complex updates behind an atomic pointer update, transforming an algorithm with non-atomic updates into one with atomic

```
 1 do {
 2
 3   /* Snapshot pointer. */
 4
 5   p = foop;
 6
 7   /*
 8    * Memory barrier on Alpha, no-op
 9    * on other CPUs.
10    */
11
12   smp_read_barrier_depends();
13
14   /* access fields. */
15
16   /* Memory barrier. */
17
18   smp_rmb();
19
20   /* Update: update pointer. */
21
22 } while (p != foop);
```

Figure 5.11: Substituting a Copy For Original Read

updates.

Note that Herlihy [42] suggests a similar pattern to switch between a pair of statically allocated structures. This approach presents some challenges when it is necessary to update the structure before all readers have dropped references to it, in which case, use of dynamic allocation as shown in Figure 5.10 can be helpful. [81, 110].

The overall effect is similar to that of the close-consistency found in the Andrew File System (AFS) [103], in that the update is invisible until the substitution, just like AFS modifications are not guaranteed to be visible until the file closes.

Example Uses: The Substitute Copy For Original pattern has been used in:

1. The Linux 2.6 kernel's System V IPC implementation, as described in Section 6.1 on Page 182.

2. The FD management patch, as described in Section 6.6 on Page 218.

In addition, a similar pattern is used in non-blocking synchronization for the same purpose, namely causing a complex update to appear to be atomic [36, 42].

5.3.4 Impose Level Of Indirection

Impose Level Of Indirection transforms an algorithm into a form to which the Substitute Copy For Original design pattern may be applied by grouping related data into one data element which may then be easily substituted.

Problem: How can an algorithm that makes complex updates to widely scattered data be modified so that the updates appear to be atomic to readers?

Context: Programs that make complex updates to widely scattered data, but where it is desired to use RCU to reduce read-side overhead and contention.

Forces:

- Not **Fresh**, since readers might still be accessing the original after the copy has been substituted.
- **Consistent**, since the copy is initialized to be consistent before being substituted.
- **Atomic**, since writes of aligned pointers are atomic.
- Does not use **Mutex**, since writes of aligned pointers are atomic.
- Does not use **Memory Barrier**, except on DEC Alpha, since there is a data dependency between accessing the pointer and dereferencing it.

Solution: Group the updated fields into a dynamically allocated data structure (or, if need be, group of data structures). This then allows the Substitute Copy For Original pattern to be applied in order to make the updates appear to be atomic.

Resulting Context: A program where updates are atomic, so that RCU may be used safely.

Design Rationale: Locality, locality, locality!

Example Uses: The Impose Level Of Indirection pattern has been considered for use in the Linux 2.6 kernel's System V IPC implementation, as discussed in Section 6.1 on Page 182. A similar pattern is used in non-blocking synchronization for the same purpose [36, 42].

5.3.5 Ordered Update With Ordered Read

Ordered Update With Ordered Read constrains the ordering of both the update and the read operations so that readers always see consistent data.

Problem: Suppose a program does not have atomic updates, but does have updates that can be made in a dependent sequence so that the results of each step of the sequence is acceptable to readers. How can this program be made safe for RCU?

Context: Program as noted above, where it is necessary to use RCU to reduce read-side overhead or contention. A dependent sequence is one in which the ordering prevents errors. For example, suppose an array and its size are stored separately, and that the array is copied to a new location with additional storage. It is an error for a reader to see the new size and the old array, because the reader could then index off the end of the old array, possibly clobbering unrelated data. All other combinations are permissible, however.

Forces:

- Not **Fresh**, since readers can see old data–only the ordering is guaranteed.
- **Consistent**, given the dependent-sequence definition of consistency.
- Not **Atomic**, since updates are made in sequence.
- Does not use **Mutex**.
- Uses **Memory Barrier** to enforce the ordering.

Solution: Force ordering of both the updates and the reads using memory barriers, as shown for the array/size example in Figure 5.12. Updaters must use some form of mutual exclusion.

```
 1 /* Array update.  */
 2
 3 newarray = kmalloc(sizeof(*newarray) * newsize);
 4 spin_lock(&array_lock);
 5 oldarray = array;
 6 for (i = 0; i < size; i++) {
 7   newarray[i] = array[i];
 8 }
 9 for (i = size; i < newsize; i++) {
10   initialize(&newarray[i]);
11 }
12 smp_wmb();
13 array = newarray;
14 smp_wmb();
15 size = newsize;
16 spin_unlock(&array_lock);
17 . . .
18
19 /* Array lookup. */
20
21 cursize = size;
22 smb_rmb();
23 curarray = array;
24 if (idx >= cursize) {
25   /* handle index out of range. */
26 } else {
27   /* access element. */
28 }
```

Figure 5.12: Ordered Update With Ordered Read

Resulting Context: Program where readers see sufficiently consistent data to avoid failures. This program may have many memory-barrier instructions, which may hamper readability. One way of avoiding this is to use Pure RCU, placing a `synchronize_kernel()` call in place of the `smp_wmb()` primitives on lines 12 and 14,[4] rendering the `smp_rmb()` unnecessary, but also substantially increasing the cost and latency of updates. Future work includes investigating a "memory-barrier shootdown" primitive, which may allow the same effect to be obtained in some circumstances at lower cost.

[4]The effect of this is to replace the memory-barrier machine instruction provided by `smp_wmb()` with the `synchronize_kernel()` primitive, which blocks waiting for a full grace period to elapse. This change would also require that the spinlocks on line 4 and 16 be replaced by sleeplocks.

Design Rationale: Memory barriers are cheaper than synchronization primitives.

Example Uses: Ordered Update With Ordered Read is used by:

1. the Linux 2.6 kernel's System V IPC implementation, described in Sections 6.1 on Page 182,
2. the Linux 2.6 kernel's directory-entry cache, described in 6.2 on Page 195, and
3. the FD management patch, described in 6.6 on Page 218.

5.3.6 Global Version Number

Global Version Number transforms an algorithm into a form where it can tolerate both stale and inconsistent data by maintaining a global version number and also associating a version number with each element. Readers can then sample the global version number before and after the access, and retry the access if there was an intervening change.

Problem: How does an algorithm using RCU ensure that a given read-only access to a data structure returns data that is both consistent and fresh?

Context: An algorithm using RCU where all updates are atomic, and where these atomic updates also atomically update a variable that can be used as a version number. If there are non-atomic updates visible to the RCU-protected readers, then these updates must be rendered atomic, perhaps by using the Substitute Copy For Original or Impose Level Of Indirection patterns described in Sections 5.3.3 and 5.3.4, respectively.

Forces:

- **Fresh** data ensured by retrying if there have been any updates during the access.
- **Consistent** data ensured by retrying if there have been any updates during the access.
- Not **Atomic**.

- Does not use **Mutex**.
- Uses **Memory Barrier** to ensure that the entire access is performed between the two checks of the global version number.

Solution: The solution is to maintain a global version number that is incremented as part of each atomic update. Two uses of Global Version Number may be found in the Linux 2.6 kernel's directory-entry cache, as discussed in Section 6.2 on Page 195.

Another approach is the non-blocking linked-list deletion algorithm presented by Cheriton [36], shown in Figure 5.13, but with memory barriers inserted as needed for machines without sequentially consistent memory models, and placed into a C function.

This function requires a special list header in which to store the version number, shown in lines 1-4 of the figure. The do-while loop retries the deletion until successful. The label on line 13 retries in case of version-number mismatch. Line 14 provides for backoff in case of high contention. Line 15 takes a snapshot of the current version number, and line 16 ensure that the operations on the following lines are in fact executed after the version number snapshot is taken. Without this memory barrier, both the compiler and the CPU would be within their rights to reorder execution. The while-loop from lines 18 to 27 searches the linked list to find the element `elt` that is to be deleted. The if-statement on line 19 would ordinarily be sufficient, but in this case, it could be fooled by concurrent list manipulations. Therefore, the version-number check on lines 21-23 is needed, as is the memory barrier on line 20, to prevent the check from being reordered. Only if the version numbers match will the return-statement on line 24 signal failure. Otherwise, line 26 advances to the next element in the list. Once the element is found, lines 28-30 attempt to atomically remove the element and increment the version number. If this fails, another pass through the loop retries. Otherwise, line 31 signals success by returning a pointer to the now-deleted element.

Resulting Context: A program that rejects stale or inconsistent data, thus being amenable to use with RCU.

```
 1 struct lookhead {
 2   struct looktab *lh_head;
 3   int lh_version;
 4 } lookhead_t;
 5
 6 struct looktab *
 7 delete(lookhead_t *list, struct looktab *elt)
 8 {
 9   struct looktab *p;
10   int version;
11
12   do {
13   retry:
14     backoffIfNeeded();
15     version=list->lh_version;
16     smp_mb();
17     p = list->lh_head;
18     while (p->next != elt) {
19       if (p == NULL) {
20         smp_mb();
21         if (version != list->lh_version) {
22           goto retry;
23         }
24         return NULL;
25       }
26       p = p->next;
27     }
28   } while (!DCAS(&(list->lh_version), &(p->next),
29            version,                   elt,
30            version+1,                 elt->next));
31   return (elt);
32 }
```

Figure 5.13: Non-Blocking Global Version Number Update

Design Rationale: The access succeeds only if there have been no updates. In the absence of updates, the data is guaranteed to be both fresh and consistent.

Warning: it is all too easy to misapply this pattern. For example, it is *not* sufficient to update the data and version number separately, as illustrated by the following sequence of events:

1. Updater increments the version number.
2. Reader accesses the version number.
3. Reader accesses the data structure.
4. Updater atomically updates the data structure.
5. Reader continues accessing the data structure.
6. Reader checks the value of the version number, finds a match, and erroneously believes that it has accessed a consistent copy of the data.
7. Updater increments the version number.

There are ways of repairing this bug, but they are beyond the scope of this dissertation.

Example Uses: Global Version Number is used by the Linux directory entry cache (dcache) described in Section 6.2 on Page 195. Here, this pattern is combined with Data Locking to mediate access to the version number. However, in this case, "global" is a bit misleading, since each dentry structure has its own "global" version number.

Global Version Number is also used in the K42 hash-table implementation described in Section 6.7 on Page 221. This is an unusual use in that the "number" is a pointer that is updated to point to a new block of memory, rather than being incremented with each new version. Nonetheless, this pointer's role is that of a version number.

Finally, as noted above, Global Version Number is used in Herlihy's non-blocking synchronization [42] and Cheriton's Cache Kernel [36]. This kernel does not feature RCU, but its non-blocking synchronization primitives require the same transformation.

5.3.7 Stall Updates

Stall Updates prevents excessive update rates from starving readers in the Global Version Number pattern by stalling updates when excessive read-side retries have been executed.

Problem: How do you prevent excessive update rates from starving readers in the Global Version Number pattern?

Context: Program using RCU in conjunction with the Global Version Number pattern.

Forces:

- **Fresh** because updates are stalled.
- **Consistent** because updates are stalled.
- Not **Atomic** unless other patterns are applied to make this so.
- Does not use **Mutex** unless acquiring a mutex is needed to stall updates. In most cases, setting a flag should suffice.
- Uses **Memory Barrier** in order to enforce ordering of global-sequence-number checks.

Solution: Have readers use a flag to stall updates in case of repeated failure, as shown in Figures 5.14 and 5.15. Another way of stalling updates is simply to wait for a grace period to expire between any pair of consecutive updates.

Resulting Context: A program that rejects stale or inconsistent data, thus being amenable to RCU, but which provides a high probability of readers' forward progress. Certainty of readers' forward progress can also be guaranteed, but this requires a more complex and costly solution.

Design Rationale: If they are getting in your way, make them stop!

```
 1 /* Global pointer that serves as version number.  */
 2
 3 struct foo *foop;
 4 . . .
 5
 6 /* Update: first allocate and fill in new value. */
 7
 8 while (stall_updates);
 9 p = kmalloc(sizeof(*p));
10 p->field1 = value1;
11 p->field2 = value2;
12
13 /* Update: issue memory barrier. */
14
15 smp_wmb();
16
17 /* Update: update pointer. */
18
19 q = foop;
20 foop = p;
21 synchronize_kernel();
22 kfree(q);
```

Figure 5.14: Stallable Update

```
 1 stalled_by_me = FALSE;
 2 for (;;) {
 3
 4   /* Snapshot pointer/version number. */
 5
 6   p = foop;
 7
 8   /* Memory barrier on Alpha. */
 9
10   smp_read_barrier_depends();
11
12   /* access fields. */
13
14   /* Memory barrier. */
15
16   smp_rmb();
17
18   /* Update: update pointer. */
19
20   if (p == foop) {
21     break;
22   }
23   if (!stalled_by_me) {
24     atomic_inc(&stall_updates);
25     stalled_by_me = TRUE;
26   }
27 }
28 if (stalled_by_me) {
29   stall_updates = FALSE;
30   atomic_dec(&stall_updates);
31 }
```

Figure 5.15: Stalling Updates

Table 5.2: Locking Design Pattern Force Index

Speedup	Contend	Ovhd	R/W	Complex	Pattern
− − −	+ + +	+ + +	− − −	+ + +	Sequential Program (5.2.2)
0	− −	0	− − −	++	Code Locking (5.2.2)
+	+	0	+	− −	Data Locking (5.2.2)
+ + +	+ + +	+?	+	?	Data Ownership (5.2.2)
++	++	++	+	− −	Parallel Fastpath (5.2.2)
++	+	+	+ + +	−	Reader/Writer Locking (5.2.2)
+ + +	++	++	+ + +	−?	**Pure RCU** (5.2.3)
+	+	++	+ + +	+	**RCU Existence Locks** (5.2.4)
+ + +	++	++	+ + +	0	**Reader-Writer-Lock/RCU Analogy** (5.2.5)
+	+	++	+ + +	++	*RCU Readers With NBS Writers* (5.2.6)
0	−	+	0	− −	Critical-Section Fusing (5.2.2)
0	+	−	0	+	Critical-Section Partitioning (5.2.2)

Example Uses: None known (but proposed for renames in dcache).

5.4 Discussion

The following two sections present an index of the locking design patterns and the RCU transformational design patterns.

5.4.1 Index to Locking Design Patterns

This section summarizes the locking design patterns with an index that compares and contrasts them, showing where each is most appropriate. The pattern listed in italics is a provisional pattern, because although it seems likely to be quite important, there is only one known use. Three uses are required for it to officially be considered a true pattern.

Table 5.2 compares how each of the patterns resolves each of the forces. Plus signs indicate that a pattern resolves a force well. For example, Sequential Program resolves Contention and Overhead perfectly due to lack of synchronization primitives, and Complexity perfectly because sequential implementations of programs are better understood and more readily available than are parallel versions.

Minus signs indicate that a pattern resolves a force poorly. Again, Sequential Program provides extreme examples with Speedup since a sequential program allows no speedup[5] and with Read-to-Write Ratio because multiple readers cannot proceed in parallel in a

[5] If you run multiple instances of a sequential program in parallel, you have used Data Locking or Data Ownership instead of Sequential Program.

Table 5.3: RCU Transformational Pattern Index

Fresh	Consistent	Atomic	Mutex	Memory Barrier	Pattern
Y	N	N	Y	Y	Mark Obsolete Objects (5.3.2)
N	Y	Y	N	n	Substitute Copy For Original (5.3.3)
N	Y	Y	N	n	*Impose Level Of Indirection* (5.3.4)
N	Y	N	N	Y	Ordered Update With Ordered Read (5.3.5)
Y	Y	N	N	Y	Global Version Number (5.3.6)
Y	Y	Y	Y	Y	*Stall Updates* (5.3.7)

sequential program.

Question marks indicate that the quality of resolution is quite variable. Programs based on Data Ownership can be extremely complex if CPUs must access each other's data. If no such access if needed, the programs can be as trivial as a script running multiple instances of a sequential program in parallel.

See the individual patterns for more information on how they resolve the forces.

5.4.2 Index to Transformational Patterns

Table 5.3 shows an index of RCU transformational patterns. The first five columns record whether the pattern resolves the corresponding force, and the last column gives the name of the pattern. This name is italicized for patterns with fewer than three uses, indicating a provisional pattern that has not yet withstood the test of time.

Entries marked with a lower-case "n" are "no" on all CPU architectures except for DEC Alpha.

These two indices assist in determining when to apply RCU to a given algorithm, and which patterns are appropriate to a given situation.

Chapter 6

Selected Applications of RCU Design Patterns

This chapter presents a few uses of RCU in the K42 and Linux operating system kernels, followed by a summary of uses of RCU in VM/XA, DYNIX/ptx, K42, and Linux. Each case identifies the patterns it uses. This author acted as architect and code reviewer for the Linux uses, and as reviewer for the K42 uses. In both cases, this author identified the patterns used. The designers and implementers are called out in each section.

RCU has enjoyed greatly increased use over the past decade:

1. Section 6.1 describes application of RCU to the Linux 2.6 kernel's System V IPC implementation, resulting in a 5% system-level improvement in a database benchmark running on a two-CPU system, and an order-of-magnitude improvement in a System V semaphore microbenchmark running on an eight-CPU system. The complexity of the modification was quite small, adding only 342 lines of code and deleting 191 for a net addition of 151 lines of code to the kernel.

2. Section 6.2 gives an overview of the use of RCU in the Linux 2.6 kernel's directory-entry cache (dcache). This modification resulted in a 26% improvement in throughput on a system-level SDET-motivated multiuser benchmark, and illustrates the use of the Global Version Number design pattern.

3. Section 6.3 covers use of RCU to enable dynamic NMI registry and unregistry in the Linux 2.6 kernel. This modification was quite simple compared to solutions based

on traditional locking or non-blocking synchronization, and in addition completely eliminated read-side synchronization overhead.

4. Section 6.4 looks at use of RCU to narrow a module-unloading race. This use illustrates the Pure RCU design pattern.

5. Section 6.5 examines incremental use of RCU to eliminate a starvation condition in the Linux 2.6 kernel's `get_pid_list()` function, which creates a list of PIDs for `/proc`. This change also resulted in a significant performance increase as measured by a SPEC SDET benchmark, and required only 13 lines added and 7 lines deleted for a six-line net addition of code to the kernel. This modification affects only this one code path; the other code paths continue to use locking. This remains a patch; it has not yet been accepted into the Linux kernel.

6. Section 6.6 describes use of RCU to allow lock-free file-descriptor lookup in the Linux 2.6 kernel. This change resulted in more than a 25% increase in throughput on the system-wide chat benchmark. However, a simpler optimization for single-threaded processes yielded even better results for that case. Although addition of RCU sped up the multithreaded case, it slowed down the more common single-threaded case compared to the simpler optimization. Work continues to construct an algorithm that speeds up the multithreaded case without slowing down the single-threaded case. This section demonstrates a case that is particularly challenging to RCU.

7. Section 6.7 shows how RCU may be combined with non-blocking synchronization to eliminate synchronization operations from read-side code and also eliminate the need for type-safe memory in a hash-table implementation in the K42 research OS. This change resulted in a significant performance increase compared to the previous lock-based implementation.

8. Section 6.8 summarizes VM/XA, Dynix/PTX, and K42 uses of RCU that were not described in the earlier sections, SuSE 7.3 Update (a Linux distribution) uses of RCU, and Linux 2.6 uses of RCU that were not described in the earlier sections.

Each section describes the patterns used, and Section 6.9 summarizes the experience and discusses the results. The patterns used by the Linux 2.6.0-test1 kernel are summarized in Table 6.1.

Table 6.1: RCU Usage in Linux 2.6.0-test1

Subsystem	RCU Pattern
System V IPC	Mark Obsolete Objects (5.3.2) Ordered Update With Ordered Read (5.3.5) RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5) Substitute Copy For Original (5.3.3)
Directory-entry cache	Global Version Number (5.3.6) Ordered Update With Ordered Read (5.3.5) RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5) Stall Updates (5.3.7)
oprofile NMI handlers	Pure RCU (5.2.3)
IP route cache	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
Module load failure races	Pure RCU (5.2.3)
tasklist patch	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
FD management patch	Ordered Update With Ordered Read (5.3.5) RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5) Substitute Copy For Original (5.3.3)
IPMI handlers	Pure RCU (5.2.3)
DECNET routing	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
SNAP protocol unregister	Pure RCU (5.2.3) RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
802.1Q VLAN	Reader-Writer-Lock/RCU Analogy (5.2.5) Pure RCU (5.2.3)
Ethernet bridge	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
Combined bridging/routing	Pure RCU (5.2.3)
Packet handler	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
IPv4/IPv6 Net filter	Pure RCU (5.2.3) RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
IPv4 and IPv6 protocol switch	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
IPv4 tunneling	RCU Existence Locks (5.2.4) Reader-Writer-Lock/RCU Analogy (5.2.5)
Raw socket protocol	RCU Existence Locks (5.2.4)

6.1 System V IPC

This section describes how RCU was used to break up the global locks used by Linux's System V IPC primitives, as described in an earlier publication by myself and others [11]. These locks guard the following: (1) mapping from IPC identifiers to corresponding `kern_ipc_perm` structures, (2) expanding the mapping arrays, and (3) individual IPC operations. A straightforward modification would replace these global locks with reader-writer locks, using the Reader/Writer Locking pattern described in Section 5.2.2 on Page 146, allowing mapping operations to be performed in parallel.

However, with the assistance of Dipankar Sarma and Maneesh Soni, Mingming Cao took the additional step of following the Reader-Writer-Lock/RCU Analogy described in Section 5.2.5 on Page 153, replacing the global locks with (1) RCU to guard the mapping arrays and (2) per-`kern_ipc_perm` locks to guard the IPC operations using the Data Locking pattern described in Section 5.2.2 on Page 145, which resulted in significant system-level speedups on database benchmarks. This modification also serves to illustrate use of the Ordered Update With Ordered Read (described in Section 5.3.5 on Page 169) and the Mark Obsolete Objects pattern (described in Section 5.3.2 on Page 161) to prevent access to stale data.

The remainder of this section focuses on the changes to the System V semaphores; analogous changes were made to message queues and shared memory.

6.1.1 Semaphore Data Structures

The semaphore data structures are shown in Figure 6.1. The global `ipc_ids` structure tracks the state of all semaphores currently in use. Among other things, it contains a global lock `ary` and a pointer `entries` that points to an array of pointers of `ipc_id` structures. Each such entry is either NULL or points to a `sem_array` structure, which represents a set of semaphores that has been created by a single `semget()` system call. The array of `ipc_id` structures is dynamically expanded as required using the Substitute Copy For Original pattern described in Section 5.3.3 on Page 163; see the discussion of the `grow_ary()` function in Section 6.1.5 on Page 188. The `sem_array` structure is allocated by

a `semget()` system call and deleted by a `semctl(IPC_RMID)` system call. The individual semaphores in a set are each represented by a `sem` structure.

Each `semop()` system call presents the `semid` for the semaphore, which must be looked up in this data structure to locate the corresponding `sem_array`. Thus, each and every semaphore operation requires that this data structure be traversed.

The `ipc_ids` field `ary` is a `spinlock_t` that protects the entire data structure. This simple locking design prevents System-V semaphore operations from proceeding in parallel. In addition, the cacheline containing the `ary` spinlock is thrashed among all CPUs.

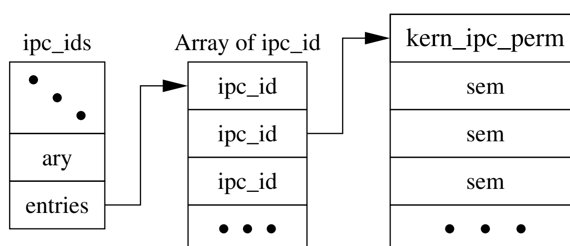


Figure 6.1: Semaphore Structures with Global Locking

Use of RCU permits fully parallel operation of different semaphores and fully parallel translation of a semaphore ID into the corresponding `sem_array` pointer. However, a few changes to the data structure are required, as shown in Figure 6.2. To begin with, the `ipc_id` array and the `sem_array` are each prefixed with an `ipc_rcu_kmalloc` structure which contains the `rcu_head` structure that RCU's `call_rcu()` function needs to track these structures during a grace period. In addition, since there is no longer a global `ary` lock, each individual `sem_array` must have its own individual `lock` to protect operations on the corresponding set of semaphores.

The final change is motivated by the fact that the translation from semaphore ID to `kern_ipc_perm` cannot tolerate the stale data that could result when an ID translation races with an `semctl(IPC_RMID)` removing that same ID. The possibility of stale data is avoided using the Mark Obsolete Objects pattern described in Section 5.3.2 on Page 161 via a `deleted` flag in the `kern_ipc_perm` structure, guarded by that structure's `lock` field. This `deleted` flag is set just after removing the corresponding `sem_array` but before

starting the grace period. The entire removal operation is performed holding the `lock` field in the `kern_ipc_perm` structure. Any attempt to lock a semaphore structure that has the `deleted` flag set then behaves as if the structure is nonexistent, as will be shown in the following sections.

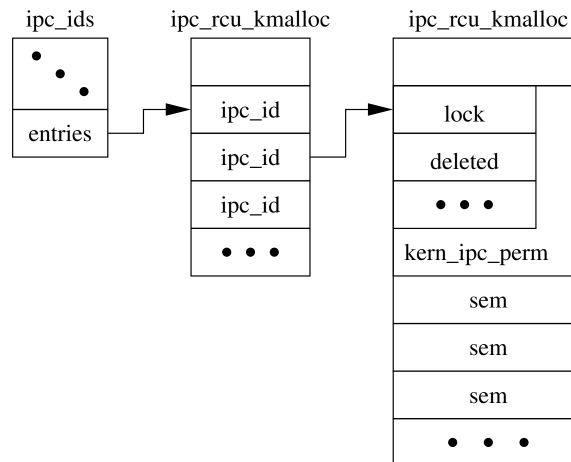


Figure 6.2: Semaphore Structures with RCU

6.1.2 Semaphore Removal

The deletion process is performed by `ipc_rmid`, as shown in Figure 6.3. This function is called with the `lock` held, and returns with it held. Lines 4-9 obtain a pointer to the `sem_array` structure. Line 10 NULLs the pointer to `sem_array`, removing any path from a persistent variable to this structure. Lines 11-12 perform a debug check, which could be triggered by locking design bugs, among other things. Lines 13-22 adjust the count of semaphores in response to the removal of this one, and then, if this semaphore had the largest ID, scans down the array of `ipc_ids` to find the new largest ID. Line 23 sets the `deleted` flag, so that the next acquisition of the lock will fail (see Section 6.1.3 below), and Line 24 returns a pointer to the newly removed semaphore. The semaphore's memory is freed up by a call to `ipc_rcu_free()` by `freeary()`, which is `ipc_rmid()`'s caller and which also performs other cleanup actions, including waking up any processes that were sleeping on the newly removed semaphore.

```
 1 struct kern_ipc_perm*
 2 ipc_rmid(struct ipc_ids* ids, int id)
 3 {
 4    struct kern_ipc_perm* p;
 5    int lid = id % SEQ_MULTIPLIER;
 6    if(lid >= ids->size)
 7       BUG();
 8
 9    p = ids->entries[lid].p;
10    ids->entries[lid].p = NULL;
11    if(p==NULL)
12       BUG();
13    ids->in_use--;
14
15    if (lid == ids->max_id) {
16       do {
17          lid--;
18          if(lid == -1)
19             break;
20       } while (ids->entries[lid].p == NULL);
21       ids->max_id = lid;
22    }
23    p->deleted = 1;
24    return p;
25 }
```

Figure 6.3: Semaphore Deletion

The `deleted` flag, once set, makes the corresponding semaphore set appear to be freed up even though it is still in memory awaiting expiration of its grace period, as will be shown in the next section.

6.1.3 Semaphore Lock Acquisition

As noted earlier, each `semop()` system call presents the `semid` for the semaphore, which must be looked up to locate the corresponding `sem_array`. In addition, the semaphore state must be locked. The `semop()` system call invokes the `ipc_lock()` kernel function to do this lookup and locking, and later invokes the `ipc_unlock()` kernel function to do the corresponding unlocking. Since the `ipc_lock()` kernel function was responsible for the lock contention that motivated use of RCU, we focus on `ipc_lock()` and the functions that it interacts with.

Since the read-side code is lock-free, nothing will prevent `ipc_lock()` from racing with `ipc_rmid()`, thus possibly gaining a reference to the structure after it is marked deleted. Figure 6.4 shows how `ipc_lock()` handles this race by checking the `deleted` field. Note

```
 1 struct kern_ipc_perm*
 2 ipc_lock(struct ipc_ids* ids, int id)
 3 {
 4    struct kern_ipc_perm* out;
 5    int lid = id % SEQ_MULTIPLIER;
 6    struct ipc_id* entries;
 7
 8    rcu_read_lock();
 9    if(lid >= ids->size) {
10       rcu_read_unlock();
11       return NULL;
12    }
13    /* barrier syncs with grow_ary() */
14    smp_rmb();
15    entries = ids->entries;
16    read_barrier_depends();
17    out = entries[lid].p;
18    if(out == NULL) {
19       rcu_read_unlock();
20       return NULL;
21    }
22    spin_lock(&out->lock);
23    /* in case ipc_rmid() just freed ID */
24    if (out->deleted) {
25       spin_unlock(&out->lock);
26       rcu_read_unlock();
27       return NULL;
28    }
29    return out;
30 }
```

Figure 6.4: Detecting Semaphore Deletion

that the `ids` argument is a pointer to the sole persistent variable, while the `out` pointer declared in Line 4 is a temporary variable. Line 5 computes the "hash" used to access the array of `ipc_ids`. Line 8 marks the beginning of the RCU read-side critical section. In preemptive kernels, this will disable preemption; in non-preemptive kernels, it does absolutely nothing other than serve as a documentation aid. Lines 9-12 check for the specified ID being out of range, returning `NULL` for failure if so. Line 14 allows for interactions with the `grow_ary()` function using the Ordered Update With Ordered Read pattern described in Section 5.3.5 on Page 169. Note that since the semaphore implementation does not use linked lists, these memory-barrier primitives must be invoked explicitly–the RCU variants of the Linux list-manipulation primitives cannot be used. Lines 15-17 obtain a stable pointer to the semaphore structure. The `read_barrier_depends()` allows for interactions with the `grow_ary()` function on multiprocessors with extremely weak memory consistency models, such as the Alpha, as described in Appendix B on Page 322. Lines 18-21 attempt to get a reference to the semaphore structure, returning `NULL` if there is no such structure (perhaps due to the specified ID no longer being valid).

Line 22 acquires the semaphore structure's `lock`. Lines 24-28 check the `deleted` flag to determine if the semaphore is being removed, and, if so, returns `NULL` to signal failure. Note that because `ipc_lock()` does not block, the normal RCU grace period prevents the semaphore structure from being freed up before `ipc_lock()` can check the `deleted` flag. Finally, Line 29 returns a pointer to the semaphore structure, having successfully translated the specified ID. Note that this function returns with the semaphore's `lock` held inside an RCU read-side critical section. The `ipc_unlock()` function therefore releases the semaphore's `lock` and then ends the RCU read-side critical section by executing a `rcu_read_unlock()`.

6.1.4 Semaphore Deferred Deletion

Since `ipc_lock()` can gain a reference to a semaphore as it is being removed, a grace period must elapse between the removal and the actual freeing of the corresponding data structures, as illustrated by Figure 6.5, which shows a simplified version of `ipc_rcu_free()` function. The actual function is more complex due to the fact that blocks of memory larger

```
1 void ipc_rcu_free(void* ptr, int size)
2 {
3    struct ipc_rcu_kmalloc *free;
4    free = ptr - sizeof(*free);
5    call_rcu(&free->rcu,
6             (void (*)(void *))kfree,
7             free);
8 }
```

Figure 6.5: Freeing a Semaphore

than a page must be freed with `vfree()` rather than `kfree()`. Line 4 computes a pointer to the beginning of the structure (see Figure 6.2), which is an `ipc_rcu_kmalloc()`, which in turn is just a wrapper around an `rcu_head` structure (see Figure 4.1). This wrapping allows more common code between the `kmalloc()` and `vmalloc()` cases. Lines 5-7 then pass to `call_rcu()` pointers to the `rcu_head` structure, to the `kfree()` function, and to the semaphore structure. The `call_rcu()` function uses the `rcu_head` structure to queue up the semaphore structure during the grace period. The actual invocation of the `kfree()` function on the semaphore structure is deferred until after the end of a subsequent grace period, as specified by the RCU Existence Locks pattern described in Section 5.2.4 on Page 151.

6.1.5 Semaphore Array Expansion

If a large number of semaphores are created, the kernel will need to expand the `ipc_id` array. Use of RCU dictates that this expansion occur in parallel with ongoing searching by `ipc_lock()`. The function `grow_ary()`, shown in Figure 6.6, implements this expansion.

Lines 8-11 do limit checking. Lines 13-21 allocate the new array, copy the old array to the first part of the new array, and initialize the remainder of the new array. Lines 22 and 23 retain the size of the old array and a pointer to it. Line 25 is a memory barrier that prevents the CPU and the compiler from reordering the array initialization with the assignment of the pointer, and is a use of the Ordered Update With Ordered Read design pattern described in Section 5.3.5 on Page 169. Any such reordering could cause other CPUs to see uninitialized segments of the array, possibly crashing (or worse!). Line 26 switches the pointer over to the new array, thus using the Substitute Copy For Original

```
 1 static int grow_ary(struct ipc_ids* ids,
 2                     int newsize)
 3 {
 4    struct ipc_id* new;
 5    struct ipc_id* old;
 6    int i;
 7
 8    if(newsize > IPCMNI)
 9       newsize = IPCMNI;
10    if(newsize <= ids->size)
11       return newsize;
12
13    new = ipc_rcu_alloc(sizeof(struct ipc_id) *
14                        newsize);
15    if(new == NULL)
16       return ids->size;
17    memcpy(new, ids->entries,
18           sizeof(struct ipc_id)*ids->size);
19    for(i=ids->size;i<newsize;i++) {
20       new[i].p = NULL;
21    }
22    old = ids->entries;
23    i = ids->size;
24
25    smp_wmb();
26    ids->entries = new;
27    smp_wmb();
28    ids->size = newsize;
29
30    ipc_rcu_free(old, sizeof(struct ipc_id)*i);
31    return ids->size;
32 }
```

Figure 6.6: Expanding the Array of Pointers to Semaphores

pattern described in Section 5.3.3 on Page 163, but any accesses at this point will recognize only the old entries as valid, since the size is still the old size. Line 27 is a memory barrier that prevents the CPU and the compiler from reordering the assignment of the size to precede the pointer assignment, again, a use of the Ordered Update With Ordered Read pattern. If such a reordering were to occur while a user was attempting to access an erroneously larger `semid`, the kernel would run off the end of the old array, again, possibly crashing. Line 28 updates the size, so that new semaphores with larger `semids` may now be accommodated. Line 30 invokes `ipc_rcu_free()`, which frees the old structure after a full grace period has elapsed. Note that `ipc_rcu_free()` returns immediately, having used `call_rcu()` to queue the old array for a later `kfree()`. Finally, Line 31 returns the new size of the array.

Note that no `deleted` flag is needed here, since the old version of the array is kept valid throughout the grace period. Any semaphore in existence at the start of the racing access that is still in existence when the racing access completes will still be correctly referenced by the old array. Note that the racing access must, by definition, complete before the grace period ends–otherwise it is not a grace period.

This tolerance of stale data is typical of ID-to-address mappings, and of routing tables as well.

Note that it would be possible to store the array's size in a structure that also includes the array itself, thus removing the need for so many memory barriers, especially on non-Alpha CPUs. This would be a use of the Impose Level Of Indirection pattern described in Section 5.3.4 on Page 168. However, this pattern would require every access to the array's size be changed, which would produce a rather large patch. Therefore, this alternative was rejected in favor of a smaller patch with a larger probability of acceptance into the Linux 2.5 kernel. If the memory barriers cause either performance or software-maintenance problems, application of the Impose Level Of Indirection pattern should be considered in the Linux 2.7 kernel effort.

6.1.6 Semaphore Operation

This section presents a graphical demonstration of how `grow_ary()`, `ipc_rmid()`, and `ipc_lock` operate. The figures in this section are abbreviated forms of Figure 6.1 and Figure 6.2. Figure 6.7 shows a system with three semaphores allocated out of a maximum of eight that could be accommodated.

The results of a concurrent `grow_ary()`, `ipc_rmid()` and creation of a new semaphore are shown in Figure 6.8, but with the additional `ipc_id` array elements omitted from the figure. At this point, a concurrent `ipc_lock()` would see semaphore 4 as being deleted (note the "D" in the diagram), and would have no way of reaching the newly created semaphore 2. The lack of visibility to semaphore 2 is legal, since this semaphore was created *after* `ipc_lock()` started execution. A subsequent `ipc_lock()` would see semaphores 0, 2, and 6, but would not newly deleted semaphore 4.

Finally, Figure 6.9 shows the state of the system after a grace period. The old `ipc_id` array has been freed, as has semaphore 4. Because the grace period has completed, there can no longer be any references either to the old array or to the now-deleted semaphore 4.

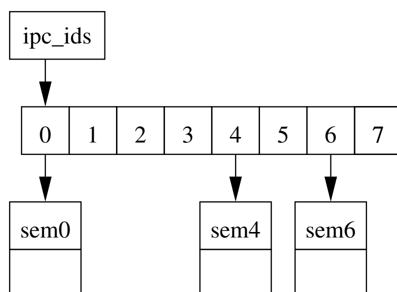


Figure 6.7: Semaphore Initial State

6.1.7 Semaphore Discussion

The RCU changes to Linux's System V IPC implementation resulted in excellent performance improvements combined with negligible increases in complexity, as is shown in the following two sections. The order-of-magnitude performance improvements make this subsystem the RCU "poster child" for RCU performance improvement.

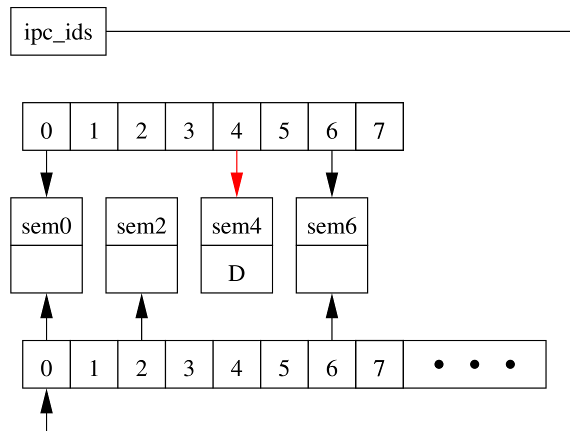


Figure 6.8: Semaphore Structures After Array Replacement

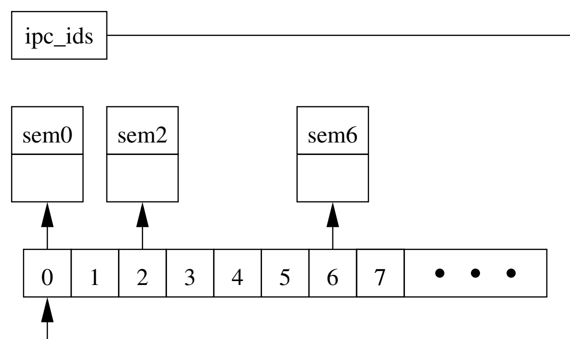


Figure 6.9: Semaphore Structures After Grace Period

Semaphore Performance

Use of RCU improves the performance of System V semaphores as measured by both system-level benchmarks and focused microbenchmarks.

The Open Source Development Lab (OSDL) used a DBT1 benchmark to evaluate system-level performance, comparing Andrew Morton's Linux 2.5.42-mm2 both with and without *ipc-rcu*. These tests were run on an Intel$^{(R)}$ dual-CPU 900MHz PIII with 256MB of memory.

The raw transaction rate for each of the five runs with each kernel are shown in Figure 6.10, which shows better than a 5% improvement due to RCU. The erratic results for the stock kernel are not unusual for workloads with lock contention. The reason for this is that if the lock contention is not too extreme, relatively deterministic workloads can "get lucky" such that multiple CPUs happen to be less likely to be contending for the same lock at the same time. As shown in Table 6.2, the difference is statistically significant: not only is *ipc-rcu*'s average three standard deviations above that of the stock kernel, but *ipc-rcu*'s smallest value of 90.4 TPS exceeds the stock kernel's median of 87.6 TPS.

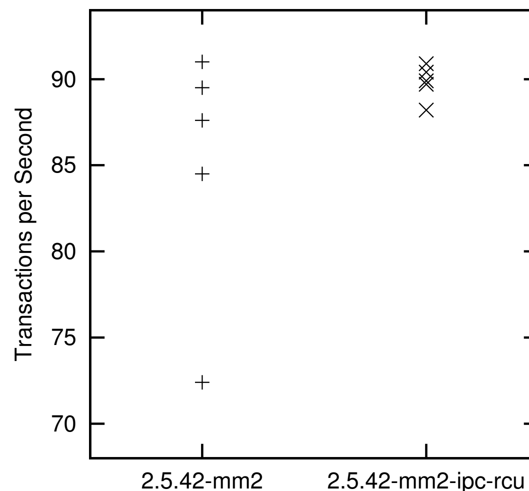


Figure 6.10: DBT1 Database Benchmark Raw Results

Table 6.2: DBT1 Database Benchmark Results (TPS)

Kernel	Average	Standard Deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

Table 6.3: semopbench Microbenchmark Results (seconds)

Kernel	Run 1	Run 2	Avg
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

Bill Hartner constructed a System V semaphore microbenchmark named semopbench and ran it on an Intel 8-CPU 700 MHz PIII system. The results in Table 6.3 clearly show the order-of-magnitude reduction in runtime obtained by applying the reader-writer-locking/RCU analogy to System V IPC mechanisms.

Semaphore Complexity

The RCU changes to the System V IPC implementations inflicted less than 5% expansion of code size, as shown in Table 6.4. This change increased the overall code size by only 151 lines. This order-of-magnitude performance benefit is well worth the modest increase in complexity.

Table 6.4: Semaphore Change in Lines of Code

				Total Lines		
	Ins/Del/Delta			New	Old	% Delta
msg.c	23	26	-3	885	888	-0.34%
sem.c	29	30	-1	1289	1290	-0.08%
shm.c	102	69	33	785	752	4.39%
util.c	178	13	165	581	416	39.66%
util.h	10	53	-43	64	107	-40.19%
Total	342	191	151	3604	3453	4.37%

Of course, the system-level performance increase is a much smaller 5.3%. On the other hand, the 151-line increase in code size is an insignificant fraction of the 11.7 million lines of code in the full kernel, and even this does not include the size of the database and other software involved in the benchmark.

6.2 Linux Directory-Entry Cache

This section describes how RCU was used to improve the scalability and performance of the Linux kernel's directory-entry cache (dcache). This use of RCU illustrates the Global Version Number, Reader-Writer-Lock/RCU Analogy, Ordered Update With Ordered Read, Data Locking, and RCU Existence Locks patterns. The material in this section is adapted from an earlier publication by myself and others [79].

Linux's directory-entry cache (dcache) maintains a partial in-memory image of the combined filesystem hierarchy. This cache permits pathname lookup to proceed without disk transfers for portions of the filesystem hierarchy that have been recently accessed, greatly increasing the performance of filesystem I/O. In order to handle mount and unmount operations easily, the Linux kernel maintains a parallel image of the mount tree in `struct vfsmount` structures.

Most operating systems use one mechanism or another to cache the filesystem hierarchy. A popular alternative is to maintain a mapping from pathnames to the associated `struct inode`. Linux's dcache approach enables it to gain the full benefit for all pathname lookups, regardless of whether the search starts at the root directory, at some process's current working directory, or at a particular per-process root directory (set by the `chroot()` system call).

6.2.1 Visual Overview of dcache

This section gives a visual overview of the dcache subsystem. There is not enough room to describe *every* aspect of the dcache subsystem, but this description will have enough detail to illuminate the RCU-related changes in the Linux 2.6 kernel. Readers desiring more detail are referred to the source code; however, reading this section should provide

good preparation for diving into the source.

This section uses the example filesystem tree in Figure 6.11 to illustrate the dcache data structures and relationships. This figure shows two filesystems, with roots "r1" and "r2", respectively. The second filesystem is mounted on directory "b", as indicated by the dashed arrow. The file (or directory) "g" has not been referenced recently, and is therefore not present in dcache, as indicated by its dashed grey box.

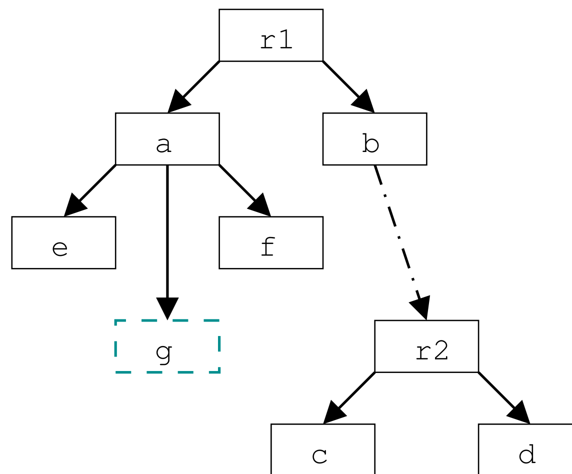


Figure 6.11: Example Filesystem Tree

The dcache subsystem maintains several views of the filesystem trees. Figure 6.12 shows the directory-structure representation. Each dentry representing a directory maintains a doubly-linked circular list headed by the `d_subdirs` field that runs through the child dentries' `d_child` fields. Each child's `d_parent` pointer references its parent. Note that the mountpoint (dentry "b") has no reference to the mounted filesystem–instead, the mountpoint's `d_mounted` flag is set, as indicated by the "(MP)" in the figure, and the mounted filesystem is looked up in the `mount_hashtable`, which will be described later.

Although one could easily imagine directly searching the filesystem view of dcache, this would be very slow for large directories. Instead, the `__d_lookup()` function computes a hash based on the parent directory's dentry pointer and the child's name, then searches the global `dentry_hashtable` for a dentry with the desired name and `d_parent`. This hash table is shown in Figure 6.13, along with the LRU list headed by `dentry_unused`.

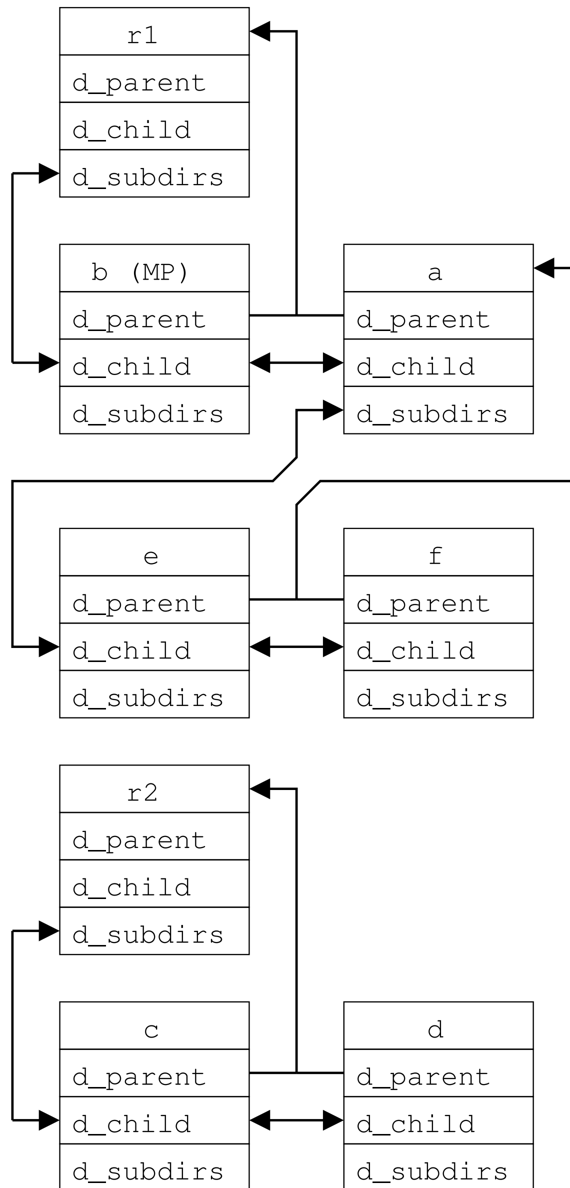


Figure 6.12: dcache Representation of Example Filesystem Tree

Note any dentry in the LRU list will usually also be in the hash table. Exceptions include cases where parent directories can time out, as can occur in distributed filesystems such as NFS.

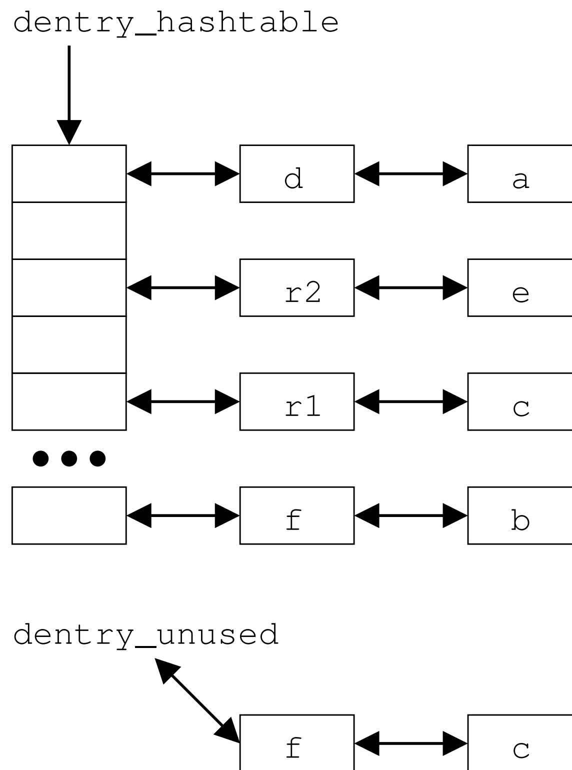


Figure 6.13: dentry Hash Table

A major purpose of the dcache subsystem is to map a pathname into an inode pointer. Hard links cause multiple pathnames to map to the same inode, and some filesystems need to be able to access all dentries referencing a given inode. Therefore, each inode maintains a list of all dentries that reference it, as shown in Figure 6.14. In addition, each dentry references its inode via the `d_inode` pointer. This `d_inode` pointer can be NULL for "negative" dentries, which lack an inode.

Negative dentries can be generated when a filesystem removes the file or directory underlying a dentry. They are also generated when someone tries to lock an non-existent file. Negative dentries can improve system performance by causing repeated accesses to a

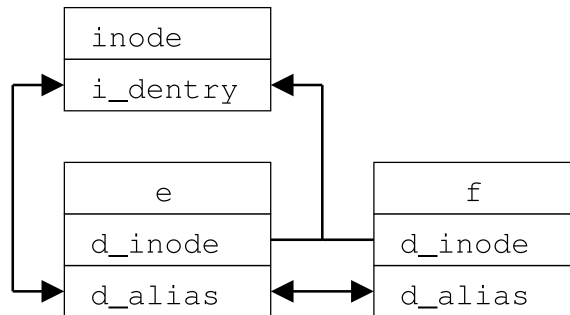


Figure 6.14: Hard-Link Alias Chains

given non-existent file to fail without needing to call into the underlying filesystem.

A high-level dentry state diagram is shown in Figure 6.15. The "normal" path through this diagram would be to:

1. Use `d_alloc()` to allocate a new dentry for a newly-referenced file, leading to state "New",

2. Use `d_add()` to associate the new dentry with its name and inode, leading to state "Hashed",

3. Use `d_put()` when done with the file, which adds the dentry to the LRU list and sets its `DCACHE_REFERENCED` bit in its `d_vfs_flags` field, leading to state "LRU Ref (Hashed)",

4. If the file is again referenced, `dget_locked()` (usually called from `d_lookup()`) will remove it from the LRU list, leading again to state "Hashed".

5. Otherwise, `prune_dcache()` will eventually remove the `DCACHE_REFERENCED` bit from the dentry's `d_vfs_flags` field, leading to state "LRU (Hashed)".

6. As before, if the file is again referenced in the "LRU (Hashed)" state, `dget_locked()` (usually called from `d_lookup()`) will remove it from the LRU list, leading again to state "Hashed".

7. Otherwise, the second consecutive call to `prune_dcache()` will invoke `d_free()` via `prune_one_dentry()`, resulting in state "Dead".

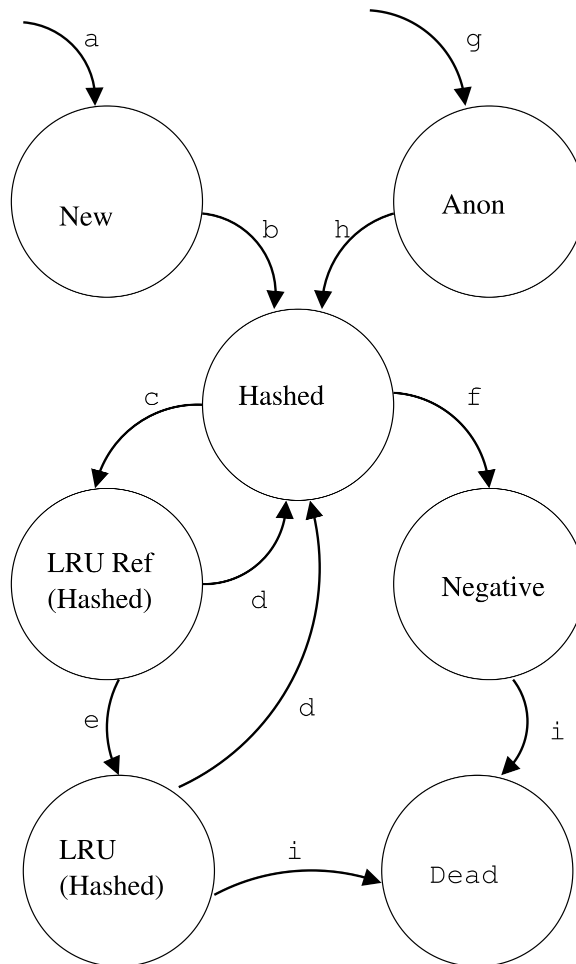
Other paths through Figure 6.15 are possible. For example, if a distributed filesystem converts a cached filehandle into a new dentry, it will invoke `d_alloc_anon()` to allocate the dentry when the corresponding object's parent is no longer represented in the dentry cache. Similarly, using `d_delete()` to delete the file or directory underlying a given dentry would move that dentry to the "Negative" state, and on last close, it would advance to "Dead".

Again, Figure 6.15 does not cover every possible dcache eventuality, but it can be very helpful when reading the source code.

Figure 6.16 shows the `mount_hashtable` data structure used to map from the mountpoint dentry (which has a non-zero `d_mounted` field) to the `struct vfsmount` of the mounted filesystem. The `mounted_hashtable` hash function combines the mountpoint dentry pointer and a pointer to the `struct vfsmount` for the filesystem containing the mount point and maps to a pointer to the mounted `struct vfsmount`. This combination of dentry pointer and `struct vfsmount` allows multiple mounts on the same mountpoint to be handled gracefully.

The example filesystem layout shown in Figure 6.11 would result in `struct vfsmount` structures as shown in Figure 6.17. The "vfs1" structure references the root dentry "r1" both as the `mnt_mountpoint` and the `mnt_root` because this filesystem is the ultimate root of the filesystem tree. The "vfs2" structure references dentry "b" as its `mnt_mountpoint` and "r2" as its `mnt_root`. Thus, when the `mount_hashtable` lookup returns a pointer to "vfs2", the `mnt_root` field may be used to quickly locate the root of the mounted filesystem.

The overall shape of the mounted filesystems is reflected in the `mnt_mount`/`mnt_child` lists. These are used by functions like `copy_tree()` while doing loopback mount, which need to traverse all the filesystems that are mounted in a particular subtree of the overall pathname namespace.




```
a  d_alloc()
b  d_add()
c  dput()
d  d_lookup() / dget_locked()
e  prune_dcache()
f  d_delete()
g  d_alloc_anon()
h  d_splice_alias()
i  d_free()
```

Figure 6.15: dentry State Diagram

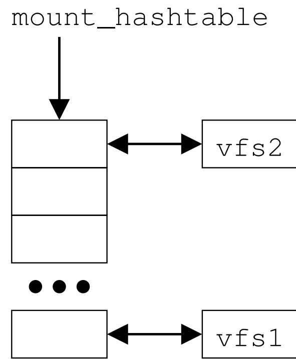


Figure 6.16: Traversing Mountpoints

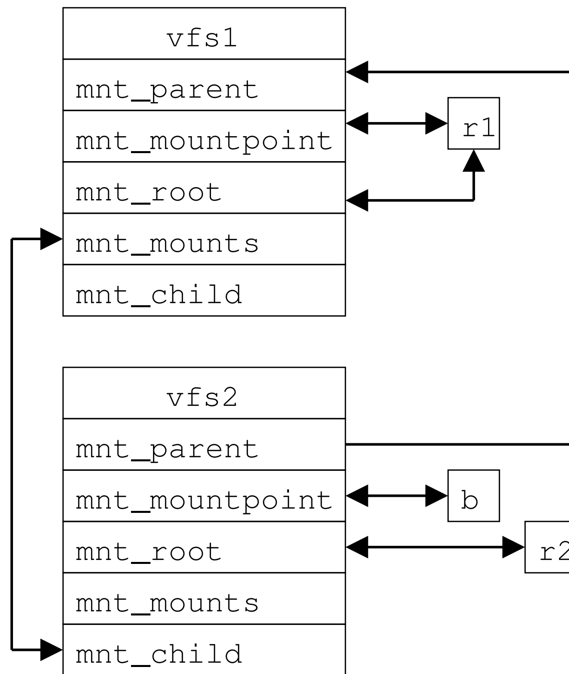


Figure 6.17: VFS Mount Tree

6.2.2 Applying RCU to dcache

The Linux 2.6 kernel is one step along the road towards RCU, in that lookups for a pathname segment within a directory are lock free, but each dentry on the path is locked and unlocked in turn. Although this is an improvement over the previous global lock, the longer-term goal would be to walk the entire path without acquiring any locks, and to perform updates in parallel.

This section describes the changes made by Dipankar Sarma and Maneesh Soni in Linux 2.6. This is not a complete description of the changes, but does provide a good view of how the dentry cache uses RCU. It should also provide a good start to people wishing to find the full story in the source code.

Pathname Segment Lookup

Pathname segment lookup is performed by the `__d_lookup()` function shown in Figure 6.18. The `__d_lookup()` function is invoked with a pointer to the parent directory's dentry and the name to be looked up. The name is passed in a `struct qstr`, which contains a pointer to the string, its length, a precomputed hash value for the dcache hash table, and a place for the name itself, if desired.

Lines 4-6 unmarshall the `struct qstr` into some local variables. Line 7 hashes the combination of the name and the parent dentry pointer into the global dcache hash table, yielding a pointer to the appropriate hash chain.

Lines 11 and 48 demark the RCU-protected segment of the code, disabling preemption in `CONFIG_PREEMPT` kernels, as specified by the Reader-Writer-Lock/RCU Analogy described in Section 5.2.5 on Page 153. Lines 12-47 loop through the elements in the selected hash chain, looking for the matching dentry. Line 17 issues a memory barrier, but only on DEC Alpha, as required by its extremely weak memory-consistency model. On other CPUs, the data dependency implied by the pointer dereference suffices.

Since this lookup acquired no locks, it is possible that it is racing with a rename system call. Such a system call could move one of the dentries to another hash chain, taking this lookup with it. Lines 19 and 20 check for this race, but are not sufficient in

and of themselves. Therefore, line 21 takes a snapshot of the number of times that the current dentry has been subjected to a rename (via the dcache `d_move()` function), as specified by the Global Version Number pattern described in Section 5.3.6 on Page 171. Line 22 is a memory barrier to ensure that the snapshot is not reordered by either the compiler or the CPU, in accordance with the Ordered Update With Ordered Read pattern described in Section 5.3.5 on Page 169.

Lines 23-26 check the name hash and the parent dentry. If either fail to match, this dentry cannot possibly be the target of our lookup. Line 27 picks up the pointer to the name structure, and line 28 executes a memory barrier, but only on Alpha. Lines 29-31 check for a non-standard name comparison, such as would be found in case-insensitive filesystems. Lines 33-36 do the full name comparison for standard filesystems. for DEC Alpha at line 41.

If execution proceeds to line 38, we have found a child dentry with matching name. Following the Data Locking pattern described in Section 5.2.2 on Page 145, that child dentry's lock is acquired on line 38.

Now, it is possible that the child dentry was renamed after the `d_move_count` snapshot was acquired on line 21. Therefore, line 39 checks the current value of `d_move_count` against the snapshot. If the check passes, the child dentry has not been renamed out from under the lookup, and lines 40-43 increment a reference count, but only if the entry is still hashed.

Line 45 releases the child dentry's lock, and line 46 breaks out of the hash-chain search loop. Line 49 returns a pointer to the child dentry, if the lookup was successful, or NULL otherwise.

Note that `__d_lookup()` failure does not mean that failure will be returned to the user process. It might well be that the file does exist, but simply has not yet been loaded into dcache.

However, this function does not protect against all rename-race hazards. One additional race is caused by the fact that dcache uses hlist rather than list for the dcache hash chains. It does this in order to save space, since hlist requires only one rather than two pointers in the list header. However, this means that hlist, unlike list, is *not* a circular list.

```
 1 struct dentry *
 2 __d_lookup(struct dentry * parent, struct qstr * name)
 3 {
 4   unsigned int len = name->len;
 5   unsigned int hash = name->hash;
 6   const unsigned char *str = name->name;
 7   struct hlist_head *head = d_hash(parent,hash);
 8   struct dentry *found = NULL;
 9   struct hlist_node *node;
10
11   rcu_read_lock();
12   hlist_for_each (node, head) {
13     struct dentry *dentry;
14     unsigned long move_count;
15     struct qstr * qstr;
16
17     smp_read_barrier_depends();
18     dentry = hlist_entry(node, struct dentry, d_hash);
19     if (unlikely(dentry->d_bucket != head))
20       break;
21     move_count = dentry->d_move_count;
22     smp_rmb();
23     if (dentry->d_name.hash != hash)
24       continue;
25     if (dentry->d_parent != parent)
26       continue;
27     qstr = dentry->d_qstr;
28     smp_read_barrier_depends();
29     if (parent->d_op && parent->d_op->d_compare) {
30       if (parent->d_op->d_compare(parent, qstr, name))
31         continue;
32     } else {
33       if (qstr->len != len)
34         continue;
35       if (memcmp(qstr->name, str, len))
36         continue;
37     }
38     spin_lock(&dentry->d_lock);
39     if (likely(move_count == dentry->d_move_count)) {
40       if (!d_unhashed(dentry)) {
41         atomic_inc(&dentry->d_count);
42         found = dentry;
43       }
44     }
45     spin_unlock(&dentry->d_lock);
46     break;
47    }
48    rcu_read_unlock();
49    return found;
50 }
```

Figure 6.18: Lock-Free Pathname Segment Lookup

It is therefore possible that a particular dentry will be renamed such that it will land in a previously empty dcache hash chain. If this happened at the right time, the `__d_lookup()` function could incorrectly return search failure.

This scenario is handled by the upper-level `d_lookup()` function, shown in Figure 6.19. Any racing renames will be detected by the `read_seqretry()` function on line 12, which checks another Global Version Number snapshotted on line 8. Since the problematic case results in spurious failure, the check is made only on NULL return from `__d_lookup()`.

```
 1 struct dentry *
 2 d_lookup(struct dentry * parent, struct qstr * name)
 3 {
 4   struct dentry * dentry = NULL;
 5   unsigned long seq;
 6
 7   do {
 8     seq = read_seqbegin(&rename_lock);
 9     dentry = __d_lookup(parent, name);
10     if (dentry)
11       break;
12   } while (read_seqretry(&rename_lock, seq));
13   return dentry;
14 }
```

Figure 6.19: Pathname Segment Lookup Rename Race Resolution

Deferred Free

The `d_free()` function follows the RCU Existence Locks pattern described in Section 5.2.4 on Page 151 as part of the Reader-Writer-Lock/RCU Analogy pattern in order to ensure that searches in `__d_lookup()` do not find themselves plowing through the freelist. This is accomplished in the `d_free()` function shown in Figure 6.20, where line 5 uses the `call_rcu()` primitive to defer the destructive actions in the `d_callback()` function until after a grace period has elapsed. The `d_callback()` function is shown in Figure 6.21; it simply frees up large names stored separately (lines 5-7), if appropriate, then frees up the dentry itself on line 8.

```
1 static void d_free(struct dentry *dentry)
2 {
3   if (dentry->d_op && dentry->d_op->d_release)
4     dentry->d_op->d_release(dentry);
5    call_rcu(&dentry->d_rcu, d_callback, dentry);
6 }
```

Figure 6.20: Deferred Free of dentry Structures

```
1 static void d_callback(void *arg)
2 {
3   struct dentry * dentry = (struct dentry *)arg;
4
5   if (dname_external(dentry)) {
6     kfree(dentry->d_qstr);
7   }
8   kmem_cache_free(dentry_cache, dentry);
9 }
```

Figure 6.21: RCU Callback Function for dentries

Rename

The `d_move()` function shown in Figure 6.22 implements the dentry-specific portion of the `rename` system call. Line 4 excludes any other tasks attempting to update dcache, and line 5 permits `d_lookup()` to determine that it has raced with a `rename`, via the Global Version Number pattern. Lines 6-12 acquire the per-dentry lock of the file being renamed and its destination, but in memory-address order to avoid deadlock scenarios. Lines 18-22 remove the entry from its old location in the dcache hash table, if it has not already been so removed.

Line 23 updates the dentry to point to its new hash bucket, line 24 adds the dentry to its destination hash bucket, in accordance with the Reader-Writer-Lock/RCU Analogy, and line 25 updates the flags to indicate that the dentry is present in the dcache hash table. Line 29 removes the target dentry (the one being `renamed` on top of) from the dcache hash table, while lines 30 and 31 divorce the moving and target dentries from their old parents.

Line 34 changes the dentry's name, and line 35 enforces ordering in accordance with Ordered Update With Ordered Read. The name change is nontrivial due to the fact that

short names are stored in the dentry itself, while longer names are stored in separately-allocated memory. Lines 36 and 37 update the name length and hash value. Lines 39-50 connect the dentry to its new parent.

Line 51 updates the `d_move_count` so that `__d_lookup()` can detect races, in accordance with the Global Version Number pattern, and lines 52-55 release the locks.

Note that, in theory, a sustained succession of rename operations that were carefully designed to leave dentries in the same directory *and* in the same hash chain could indefinitely stall horribly unlucky lookups. One way that this could happen is if the lookup was searching for the last element in the hash chain, and that the second-to-last element was consistently renamed (thus moved to the head of the list), just as the lookup got to it. In practice, dcache hash chains are short and renames are slow. However, if this becomes a problem, it is a good candidate for the Stall Updates pattern described in Section 5.3.7 on Page 175.

6.2.3 dcache Discussion

Although this change was relatively small, it had ramifications due to the fact that there had not been a well-defined API for filesystems to interact with dcache. This resulted in a large number of bugs in the Linux 2.5 kernel due to filesystems maintainers attempting to manipulate dcache in the traditional style. Given that there is now a somewhat more formal API, it is hoped that future changes will be somewhat less traumatic.

Figure 6.23 shows the improved performance of up to 26% on an SDET-motivated multiuser benchmark running on a Linux 2.5.59 kernel patched to use RCU in the directory-entry cache over that on an unpatched 2.5.59 kernel. These benchmarks were run on a 16-CPU NUMA-Q with 700MHz PIII Intel Xeon CPUs.

Adding RCU to dcache also resulted in a 12% improvement in SPECweb99 throughput on an 8-CPU PIII Xeon server, increasing SPECweb99 throughput from 2258 to 2530 in a test that applied the *dcache_rcu* patch to the Linux 2.4.17 kernel [102].

Applying the same change to a Linux 2.5.40-mm2 kernel resulted in more than a 10% reduction in system time consumed by a build of the Linux kernel on a NUMA-Q system with sixteen 700MHz PIII CPUs, from 47.548 CPU seconds to 42.498 CPU seconds. A

```
 1 void
 2 d_move(struct dentry *dentry, struct dentry *target)
 3 {
 4   spin_lock(&dcache_lock);
 5   write_seqlock(&rename_lock);
 6   if (target < dentry) {
 7     spin_lock(&target->d_lock);
 8     spin_lock(&dentry->d_lock);
 9   } else {
10     spin_lock(&dentry->d_lock);
11     spin_lock(&target->d_lock);
12   }
13
14   /*
15    * Move the dentry to the target hash queue,
16    * if on different bucket
17    */
18   if (dentry->d_vfs_flags & DCACHE_UNHASHED)
19     goto already_unhashed;
20   if (dentry->d_bucket != target->d_bucket) {
21     hlist_del_rcu(&dentry->d_hash);
22 already_unhashed:
23     dentry->d_bucket = target->d_bucket;
24     hlist_add_head_rcu(&dentry->d_hash, target->d_bucket);
25     dentry->d_vfs_flags &= ~DCACHE_UNHASHED;
26   }
27
28   /* Unhash the target: dput() will then get rid of it */
29   __d_drop(target);
30
31   list_del(&dentry->d_child);
32   list_del(&target->d_child);
33
34   switch_names(dentry, target);
35   smp_wmb();
36   do_switch(dentry->d_name.len, target->d_name.len);
37   do_switch(dentry->d_name.hash, target->d_name.hash);
38
39   if (IS_ROOT(dentry)) {
40     dentry->d_parent = target->d_parent;
41     target->d_parent = target;
42     INIT_LIST_HEAD(&target->d_child);
43   } else {
44     do_switch(dentry->d_parent, target->d_parent);
45     list_add(&target->d_child,
46        &target->d_parent->d_subdirs);
47   }
48
49   list_add(&dentry->d_child,
50      &dentry->d_parent->d_subdirs);
51   dentry->d_move_count++;
52   spin_unlock(&target->d_lock);
53   spin_unlock(&dentry->d_lock);
54   write_sequnlock(&rename_lock);
55   spin_unlock(&dcache_lock);
56 }
```

Figure 6.22: Renaming dentries

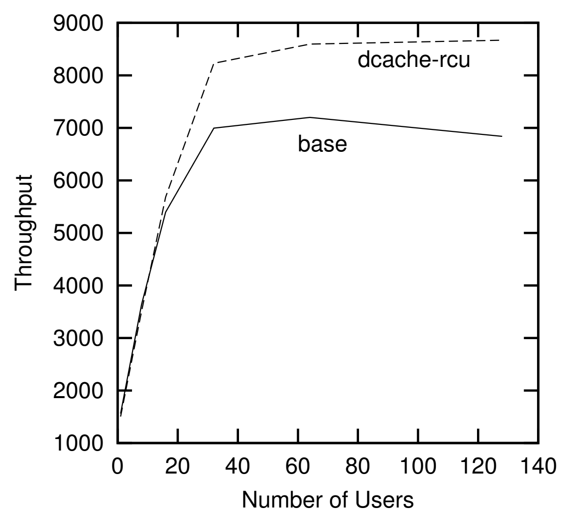


Figure 6.23: Multiuser Benchmark Performance

similar test run on a uniprocessor 700MHz P3 Xeon system, but based on the Linux 2.5.42 kernel, showed no change. Therefore dcache RCU not only increases scaling for high-end machines, it also does no harm to performance on low-end machines [101], providing a statistically insignificant performance increase for 2- and 4-CPU machines.

This is an especially impressive result, given the dcache subsystem's reputation for complexity and the large number of filesystems that depend on dcache, and also given that the change added only 196 lines and deleted 70, for a net addition of only 126 lines of code to the kernel. Future work includes modifying dcache to do lockfree traversal of full paths rather than just path segments, and on supporting fully parallel updates.

6.3 RCU Synchronizing With NMIs

This section is adapted from one of this author's prior publications [71].

Retrofitting existing code with RCU as shown above can produce significant performance gains, but of course the best results are obtained by designing RCU into the algorithms and code from the start.

The i386™ oprofile code contains an excellent example of designed-in RCU, using the Pure RCU pattern described in Section 5.2.3 on Page 147. This code was written by John Levon based on a patch by Zwane Mwaikambo, with guidance by Dipankar Sarma. This code can use NMIs (non-maskable interrupts) to do profiling independently of the normal clock interrupt, thereby permitting profiling of the clock interrupt handler. Synchronizing with NMIs has traditionally been difficult, since there is by definition no way to block an NMI. Straightforward locking designs are therefore subject to deadlock, where the CPU holding the lock receives an NMI, and the NMI handler spins forever on this same lock. Another approach is to effectively mask NMIs in software using things like `spin_trylock()`, but this incurs cache-bouncing and memory-barrier overhead, and the NMIs thus masked are lost.

The solution in `nmi_timer_int.c` is as shown in Figure 6.24.

The `synchronize_kernel()` ensures that any NMI handlers that were executing the

```
static void timer_stop(void)
{
        enable_timer_nmi_watchdog();
        unset_nmi_callback();
        synchronize_kernel();
}

static struct oprofile_operations nmi_timer_ops = {
        .start = timer_start,
        .stop = timer_stop,
        .cpu_type = "timer"
};
```

Figure 6.24: NMI Timer Stop Function

old NMI callback upon entry to `timer_stop()` have completed before `timer_stop()` returns. The code for `oprofile_stop()` and `oprofile_shutdown()` shown in Figure 6.25 illustrates why this is important. Note that `oprofile_ops->stop()` invokes `timer_stop()`. Therefore, if `oprofile_stop()` and `oprofile_shutdown()` were called in quick succession, the newly freed CPU buffers could be accessed by an ongoing NMI, which could surprise any code quickly reallocating this memory.

```
void oprofile_stop(void)
{
        down(&start_sem);
        if (!oprofile_started)
                goto out;
        oprofile_ops->stop();
        oprofile_started = 0;
        /* wake up the daemon to read what remains */
        wake_up_buffer_waiter();
out:
        up(&start_sem);
}

void oprofile_shutdown(void)
{
        down(&start_sem);
        sync_stop();
        if (oprofile_ops->shutdown)
                oprofile_ops->shutdown();
        is_setup = 0;
        free_event_buffer();
        free_cpu_buffers();
        up(&start_sem);
}
```

Figure 6.25: oprofile Shutdown Code

Use of RCU eliminates this race very naturally, without incurring any locking or

memory-barrier overhead. This race would be very difficult to resolve using locking, which may explain why very few operating systems have dynamically changeable NMI handlers.

6.4 RCU and Module Race Reduction

The material in this section is adapted from an earlier publication by myself and others [78].

Linux 2.4 is subject to races between module unloading and use of that module, due to the fact that a module user must gain a reference to that module before announcing its presence, and the module might be unloaded in the meantime. These races can result in the racing code that is attempting to use the module holding a reference to newly freed memory, most likely resulting in a failure (or "oops", in Linux parlance).

One way to reduce the likelihood of these races occurring is to wait for a grace period after removing the module structure from the `module_list` before `kfree()`ing it in `free_module()`, applying the Pure RCU pattern described in Section 5.2.3 on Page 147. As long as the module user was not preempted between the time it obtains the reference and announces its presence, it is guaranteed to find a valid module data structure. Races can still occur,[1] but the race's window has been decreased substantially. The change is a one-liner (not counting comments), as shown in Figure 6.26.

As noted earlier, this change does not address all the module-unloading problems. However, perhaps it can be a basis for a full solution–Rusty Russell rewrote the modules subsystem in the Linux 2.6 kernel, but some module-unload issues remain. The approach described in this section was based on an earlier suggestion by Rusty Russell, and added to SuSE Linux by Andi Kleen.

6.5 Incremental Use of RCU on tasklist Locking

This section is adapted from one of this author's prior publications [71].

Use of RCU is not an all-or-nothing affair. RCU may be applied incrementally to particular code paths as needed. A good example of this is a patch coded by Dipankar

[1] For example, if the module user is preempted, or in cases where the module user avoids announcing its presence.

```
 1 @@ -1065,6 +1066,12 @@
 2          p->next = mod->next;
 3      }
 4      spin_unlock_irqrestore(&modlist_lock,
 5                             flags);
 6
 7 +   /* Wait for all other cpus to go
 8 +    * through a context switch. This
 9 +    * doesn't plug all module unload
10 +    * races, but at least some of
11 +    * them and makes the window much
12 +    * smaller.
13 +    */
14 +   synchronize_kernel();
15
16          /* And free the memory.  */
```

Figure 6.26: Module Unloading

Sarma that prevents `ls /proc` from blocking `fork()`.

The problem is that `get_pid_list()` traverses the entire tasklist in order to build the PID list needed by `ls /proc`. It read-holds `tasklist_lock` during this traversal, blocking updates to the tasklist, such as those performed by `fork()`. On machines with large numbers of tasks, this can cause severe difficulties, particularly given multiple instances of certain performance-monitoring tools.

Dipankar's modifications are shown in Tables 6.5 and 6.6, changing only two files, adding thirteen lines and deleting seven for a six-line net addition to the kernel, deleting a pair of `tasklist_lock` uses. None of the other 249 uses of `tasklist_lock` are modified.

The changes make use of the Reader-Writer-Lock/RCU Analogy and RCU Existence Locks patterns described in Sections 5.2.5 on Page 153 and 5.2.4 on Page 151, respectively, and are as follows:

1. The `read_lock()` and `read_unlock()` of `tasklist_lock` in `get_pid_list()` are replaced by `rcu_read_lock()` and `rcu_read_unlock()`, respectively.

2. A `struct rcu_head` is added to `task_struct` in order to track the task structures waiting for a grace period to expire.

3. The `put_task_struct()` macro invokes `__put_task_struct()` via `call_rcu()` rather than directly, ensuring that all concurrently executing `get_pid_list()` invocations

complete before any task structures that they might have been referencing are freed. This is an example of the RCU Existence Locks pattern.

4. The `SET_LINKS()` and `REMOVE_LINKS()` macros make use of the `_rcu()` forms of the list-manipulation primitives.

5. The `for_each_process()` macro gets a `read_barrier_depends()` to make this code safe for the DEC Alpha.

This example demonstrates use of RCU for a late-in-cycle optimization.

The task-list patch is an 85-line context diff that adds a net six lines to the kernel, broken down as shown in Figure 6.27. Despite the small size of this change, it yields significant system-level performance benefits, as shown in Figure 6.28. This data was collected on a machine with 16 Pentium III Xeon processors running at 700MHz and 32 GB of memory.

Lines		
Added	Deleted	Reason
2	0	Include rcupdate.h
2	2	Read-side locking
6	2	Add `call_rcu()` to defer destruction
2	2	Convert to RCU list macros
1	1	Add Alpha-only memory barrier
13	7	Total

Figure 6.27: Task-List RCU Patch

This example demonstrates one of the great strengths of RCU in general and of the Reader-Writer-Lock/RCU Analogy pattern in particular, namely, that they can be applied incrementally to an existing design. This change prevents the `get_pid_list()` code path (invoked by "`ls /proc`" commands) from starving other manipulations of the task list, such as `fork()` and `exec()`. This `get_pid_list()` code path becomes lock free, the update path uses `call_rcu()` to defer destruction, and all other code paths are unaffected.

Table 6.5: Applying RCU to get_pid_list()

Original Code	RCU Version

<pre>1 static int get_pid_list(int index,
2   unsigned int *pids)
3 {
4   struct task_struct *p;
5   int nr_pids = 0;
6
7   index--;
8   read_lock(&tasklist_lock);
9   for_each_process(p) {
10   int pid = p->pid;
11   if (!pid_alive(p))
12   continue;
13   if (--index >= 0)
14   continue;
15   pids[nr_pids] = pid;
16   nr_pids++;
17   if (nr_pids >= PROC_MAXPIDS)
18   break;
19   }
20   read_unlock(&tasklist_lock);
21   return nr_pids;
22 }</pre>
	<pre>1 static int get_pid_list(int index,
	2 unsigned int *pids)
	3 {
	4 struct task_struct *p;
	5 int nr_pids = 0;
	6
	7 index--;
	8 rcu_read_lock();
	9 for_each_process(p) {
	10 int pid = p->pid;
	11 if (!pid_alive(p))
	12 continue;
	13 if (--index >= 0)
	14 continue;
	15 pids[nr_pids] = pid;
	16 nr_pids++;
	17 if (nr_pids >= PROC_MAXPIDS)
	18 break;
	19 }
	20 rcu_read_unlock();
	21 return nr_pids;
	22 }</pre>

Table 6.6: Applying RCU to get_pid_list() Helper Macros

Original Code

```
1 #define put_task_struct(t) \
2 do { \
3   if (atomic_dec_and_test(&(t)->usage)) \
4     __put_task_struct(t); \
5 } while(0)
```

RCU Version

```
1 void put_task_struct(struct task_struct *t)
2 {
3   if (atomic_dec_and_test(&t->usage))
4     call_rcu(&t->rcu,
5       (void (*)(void *))__put_task_struct,
6       t);
7 }
```

Original Code

```
 1 #define REMOVE_LINKS(p) do { \
 2   if (thread_group_leader(p)) \
 3     list_del_init(&(p)->tasks); \
 4   remove_parent(p); \
 5   } while (0)
 6
 7 #define SET_LINKS(p) do { \
 8   if (thread_group_leader(p)) \
 9     list_add_tail(&(p)->tasks, \
10                   &init_task.tasks); \
11   add_parent(p, (p)->parent); \
12   } while (0)
```

RCU Version

```
 1 #define REMOVE_LINKS(p) do { \
 2   if (thread_group_leader(p)) \
 3     list_del_rcu(&(p)->tasks); \
 4   remove_parent(p); \
 5   } while (0)
 6
 7 #define SET_LINKS(p) do { \
 8   if (thread_group_leader(p)) \
 9     list_add_tail_rcu(&(p)->tasks, \
10                       &init_task.tasks); \
11   add_parent(p, (p)->parent); \
12   } while (0)
```

Original Code

```
1 #define for_each_process(p) \
2   for (p = &init_task; \
3        (p = next_task(p)) != &init_task; )
```

RCU Version

```
1 #define for_each_process(p) \
2   for (p = &init_task; \
3        (p = next_task(p)), \
4         ({ read_barrier_depends(); 0;}), \
5         p != &init_task; \
6       )
```

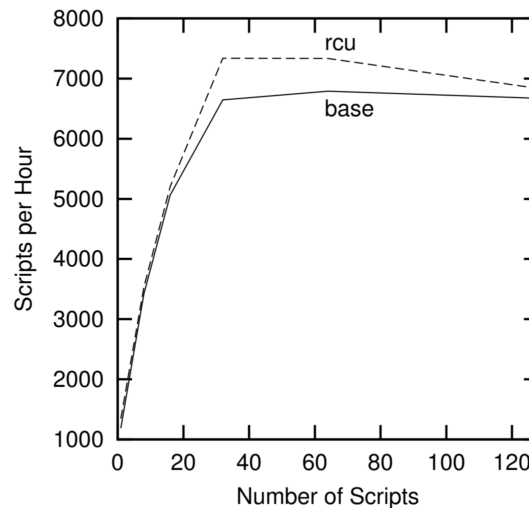


Figure 6.28: SPEC SDET Performance of Task-List RCU

Since only 20 lines needed to be changed (13 added, 7 deleted), this change is the RCU "poster child" for large benefits from tiny changes.

6.6 Scalable FD Management

This section is adapted from a prior publication by myself and others [73].

The Linux kernel's FD management maintains the data structures that map from a file descriptor to the corresponding `struct file`. This mapping is implemented as a set of arrays (pointed to by `fd`, `close_on_exec`, and `open_fds`), which can grow as the process opens more files. This section describes how RCU was applied to this subsystem.

The current FD management code uses a reader-writer lock (`file_lock`) to guard the `files_struct` state, in particular, the `fd`, `close_on_exec`, and `open_fds` pointers. Dipankar Sarma and Maneesh Soni replaced the reader-writer `file_lock` with a spinlock, deleted the `read_lock()` calls, and replaced `write_lock()` calls with `spin_lock()`. This follows the Reader-Writer-Lock/RCU Analogy pattern discussed in Section 5.2.5 on Page 153.

The `expand_fd_array()` and `expand_fdset()` functions are then cast into RCU form,

```
 1 if (i) {
 2    memcpy(new_openset, files->open_fds,
 3           files->max_fdset/8);
 4    memcpy(new_execset, files->close_on_exec,
 5           i * sizeof(struct file *));
 6    memset(&new_openset->fds_bits[i], 0, count);
 7    memset(&new_execset->fds_bits[i], 0, count);
 8 }
 9 nfds = xchg(&files->max_fdset, nfds);
10 new_openset = xchg(&files->open_fds,
11     new_openset);
12 new_execset = xchg(&files->close_on_exec,
13     new_execset);
14 write_unlock(&files->file_lock);
15 free_fdset(new_openset, nfds);
16 free_fdset(new_execset, nfds);
17 write_lock(&files->file_lock);
```

Figure 6.29: Expanding FD Array

with the update split into two phases separated by a grace period.

The original form of the update portion of `expand_fd_array()` is shown in Figure 6.29. In the RCU version, lines 1 through 13 are executed in the first phase, and lines 15 and 16 are executed after a grace period, using the `synchronize_kernel()` function to defer execution of the `free_fdset()` functions. This approach allows any tasks running on other CPUs that are still referencing the arrays pointed to by the old values of `fd`, `close_on_exec`, and `open_fds` to continue normally, in accordance with the Substitute Copy For Original pattern described in Section 5.3.3 on Page 163.

A RCU version is shown in Figure 6.30. This code must install the new arrays before updating `max_fdset`, since read-side critical sections are no longer excluded when running this code. The `smp_wmb()` calls are needed to maintain memory ordering on CPUs with extremely weak memory consistency. The `expand_fdset()` function is modified in a similar fashion. Both these changes follow the Ordered Update With Ordered Read pattern described in Section 5.3.5 on Page 169.

This patch uses a slightly different approach from that shown in Figure 6.30. Rather than using `synchronize_kernel()`, it registers RCU callbacks, which asynchronously invoke auxiliary functions to free the memory after the grace period expires. This more-complex approach is necessary for good performance, as the `synchronize_kernel()` approach results in extra context switches, whose overhead overwhelms RCU's performance.

```
 1 if (i) {
 2    memcpy(new_openset, files->open_fds,
 3           files->max_fdset/8);
 4    memcpy(new_execset, files->close_on_exec,
 5           i * sizeof(struct file *));
 6    memset(&new_openset->fds_bits[i], 0, count);
 7    memset(&new_execset->fds_bits[i], 0, count);
 8 }
 9 smp_wmb();
10 new_openset = xchg(&files->open_fds,
11     new_openset);
12 new_execset = xchg(&files->close_on_exec,
13     new_execset);
14 smp_wmb();
15 nfds = xchg(&files->max_fdset, nfds);
16 write_unlock(&files->file_lock);
17 synchronize_kernel();
18 free_fdset(new_openset, nfds);
19 free_fdset(new_execset, nfds);
20 write_lock(&files->file_lock);
```

Figure 6.30: RCU Expanding FD Array

Both of these variants are examples of the RCU Existence Locks pattern discussed in Section 5.2.4 on Page 151.

Figure 6.31 on Page 221 shows the performance benefits of the RCU version of FD management on the chat benchmark with rooms=20 and messages=500 in a 2.4.2 SMP kernel. These runs used a 1-way, 2-way, 3-way, and a 4-way PIII Xeon 700MHz system with 1MB L2 cache and 1GB of RAM. The RCU version attains over 30% more throughput at four CPUs, which should benefit all multithreaded applications that do heavy disk or network I/O. In addition, this change does not penalize uniprocessor kernels, instead showing a statistically insignificant performance increase (0.65%). In all cases, kernprof measurements revealed greatly reduced hits in the `fget()` function. Since there was no sign of heavy contention on the lock used in this code, it is probable that the increased throughput was due to reduced cache thrashing.

This change has not yet been included in the Linux 2.6 kernel. Instead, a simpler modification that greatly reduced the locking overhead of single-threaded processes has been adopted. The locking bottleneck still exists, but only for multithreaded processes. However, the change described in this section degrades performance for single-threaded processes, given that the aforementioned simple modification has been applied. Future

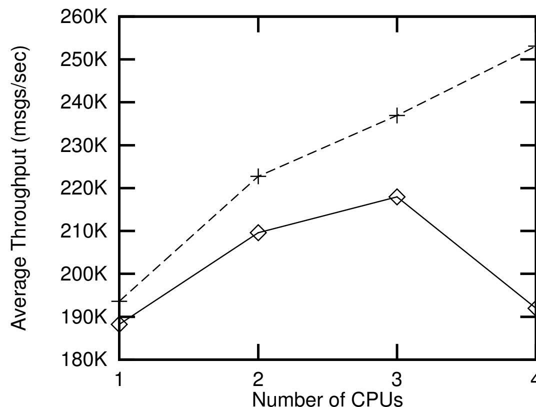


Figure 6.31: FD Management Performance

work includes investigating the feasibility of an RCU patch that does not degrade single-threaded performance.

6.7 K42 Hash Tables

Much groundbreaking work has been done in the area of non-blocking synchronization, resulting in algorithms that are both scalable and robust against process-death failures. Nonetheless, because general non-blocking synchronization algorithms require structure-reuse checks, these algorithms have seen no significant practical use.

However, deferring destruction of data structures removes the requirement both for structure-reuse checks and for read-only accesses to perform expensive atomic writes to shared storage. This deferred destruction can be accomplished by using RCU to delay destruction for one grace period. This section gives an extended description of a plug-in hash table that Marc Auslander designed and coded for the K42 operating system.

Hash tables are a fundamental data structure used to provide fast lookup. Lock contention on hash tables can severely limit scalability in multiprocessor applications. We illustrate how RCU may be used in conjunction with non-blocking synchronization to provide highly scalable hash tables, while avoiding the need for the reuse checks traditionally required by non-blocking synchronization.

This discussion assumes the standard chained hash-table implementation in which a key is "hashed" to an index. The index selects a hash-chain header or "bucket", which contains a linked list of elements representing keys with the same hash value. Each element contains a datum that is associated with the key, so that the hash table implements a mapping from key to datum.

One approach to increase scalability is to use separate locks for each bucket. This reduces contention if key lookups are well-distributed across the buckets, but not if there are a few hot keys.

Another approach is to implement a non-blocking hash table. Of course, non-blocking techniques reduce contention, but do not address communication cache misses. However, combining non-blocking techniques with RCU *can* reduce communication cache misses by eliminating them among read-only accesses. These reductions of contention and communication cache misses, when combined with elimination of priority inversions and other locking issues, can provide sufficient performance and scalability for many situations. This combination of techniques is an example of the RCU Readers With NBS Writers pattern from Section 5.2.6 on Page 158.

The fundamental operations on a hash table are lookup, insert, and remove.

The lookup operation takes a key, and either returns the corresponding datum or an error if the key is not present. There are many variations on this theme that do not change the fundamental algorithm, for example, some implementations return the address of the element associated with the key.

The insert operation inserts an element that maps the specified key to the specified datum, or an error if the key is already present. Again, there are many variations on this theme, including permitting multiple instances of the same key, or updating an existing datum if the specified key is already present. This operation marks each new element with a "valid" indication.

The remove operation deletes an element that maps the specified key to a datum, relying on the "valid" indication.

All of these operations can be viewed in terms of their effect on the single bucket that the specified key hashes to. Implementations are normally designed to operate best when

the average hash-chain length is small, in fact hash tables are frequently tuned to achieve an average chain length of one.

The lookup operation traverses the hash chain, returning the datum found in the first element with a matching key that is marked “valid”. If no such element is found, the lookup operation returns an error.

The insert operation always inserts new elements at the head of the hash chain. This can be implemented with the well-known non-blocking list-push operation, in which atomic instructions such as compare-and-swap are used to push the new element onto the hash chain. However, we must correctly handle the case where two concurrent insert operations are attempting to insert the same new key. Exactly one of these operations must succeed; the other must fail. In addition, if the key is an old key that is already present in the hash table, both operations must fail.

These behavior constraints are met by recording the value of the hash-chain header, then searching the list for a matching key. If the key is found in an element marked “valid”, the insert operation fails and an error is returned. Otherwise, the insert operation performs a compare-and-swap push using the recorded value of the hash-chain header. If any elements have been inserted during the search, or if the first element has been deleted during the search, the compare-and-swap push will fail. This is an example of the Global Version Number pattern from Section 5.3.6 on Page 171, where the header pointer acts as the global version number that changes only for insertions and removals of the head of the list. Failure is guaranteed because deferred destruction prevents a deleted element from being re-inserted into the list until after all concurrently executing insertions have completed. This guarantee is provided through use of the RCU Existence Locks pattern from Section 5.2.4 on Page 151.

The remove operation relies on the “valid” indication that is associated with each element on the hash chain. Remove searches the chain, and, if it finds a matching element marked “valid”, it uses an atomic operation such as compare-and-swap to mark the element “invalid”, as denoted by the hollow arrow from element A in step (2) of Figure 6.32, indicating that element A is now invalid through use of the Mark Obsolete Objects pattern from Section 5.3.2 on Page 161. This atomic operation must eventually succeed, since new

elements are added only to the beginning of the list, and there are only a finite number of elements following this one that may be deleted. If the remove operation reaches the end of the chain, it returns an error.

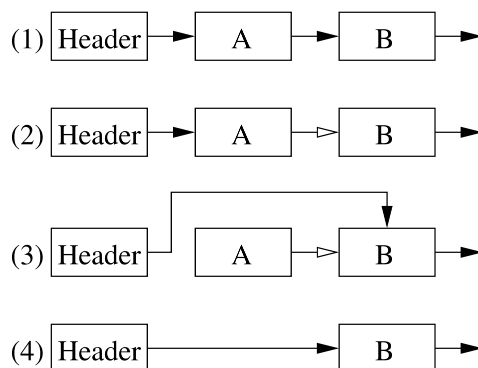


Figure 6.32: Lock-Free Hash Remove Operation

Otherwise, the remove operation must now remove the invalidated element A from the hash chain, as shown in step (3) of the figure. Although it is impossible to remove an arbitrary element from a list using non-blocking operations in the face of other concurrent removal operations, in this case, it *is* possible to remove the first invalid element, whose predecessor must be either a valid element or the hash-chain header [86]. We accomplish this by coding the "valid" indicator into the low-order bits of the "next" pointer, again, as denoted by the hollow arrows in the figures. Since removing an element involves replacing its predecessor's "next" pointer with its own "next" pointer, concurrent removal of the predecessor will change the value of the predecessor's "next" pointer, which will in turn cause the compare-and-swap operation that updates this pointer to fail. If the compare-and-swap operation succeeds, the newly removed element A's destruction (and subsequent reuse) must be deferred for a full grace period to allow any concurrent hash-chain traversals to give up any references to this element, and to ensure that the insert operation will operate correctly when an element that has been removed is reused and reinserted into the list, as shown in step (4) of the figure. The combination of the "valid" indication with the "next" pointer, the removal only of the first invalid element, and the deferred destruction are sufficient to guarantee that the remove operation is correct. An example failure-retry

scenario due to a race with insertion of a new element C is shown in Figure 6.33.

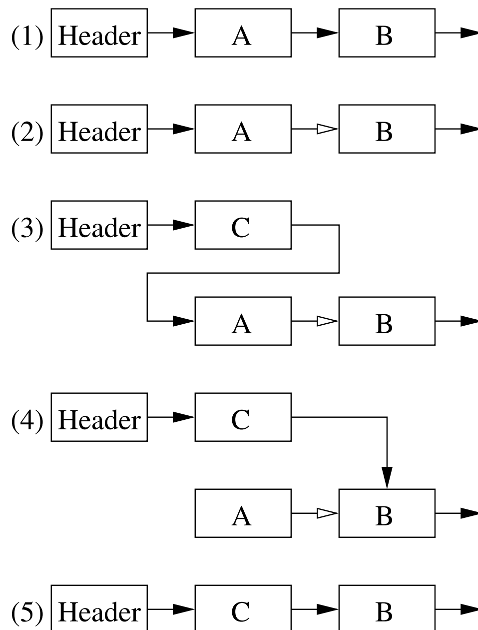


Figure 6.33: Lock-Free Hash Remove Operation with Race

Note that this algorithm can in theory result in starvation if new threads continuously add, invalidate, and try to remove elements. This starvation can be reduced by requiring that the insert operation remove any "invalid" elements from the head of the hash chain before inserting the new element.

In practice, we have instead chosen to avoid the starvation scenario described above by only removing singleton elements with compare-and-swap operations, and then using a per-bucket lock to serialize all removals from hash chains having more than one element. In a properly-tuned hash table, removals from long hash chains is so rare that the lock overhead is insignificant. We are nonetheless investigating an improved non-blocking removal algorithm.

Michael [87, 86] has recently advanced an alternative approach based solely on non-blocking synchronization that does away with the traditional requirement for reuse checks and type-safe memory. However, the RCU approach eliminates the need for expensive writes to shared memory and associated memory barriers for read-only accesses to the

hash table, which is especially beneficial to read-mostly hash tables, such as those used to implement network routing tables.

Figure 6.34 on Page 226 shows the performance benefits of the lock-free hash table when running the PostMark benchmark on a 24-CPU PowerPC system. The offered load is increased with increasing numbers of CPUs, so that the 24-CPU runs process 24 times the work as the single-CPU runs. Beyond 10 CPUs, the overhead of locking becomes quite erratic, with the outlier points caused by stable convoys of processes that can form on the hash-table lock. In contrast, the runtime required by the lock-free RCU-based hash table is quite predictable, and ranges from 2 to 10 times faster than the locked hash table when running on 24 CPUs.

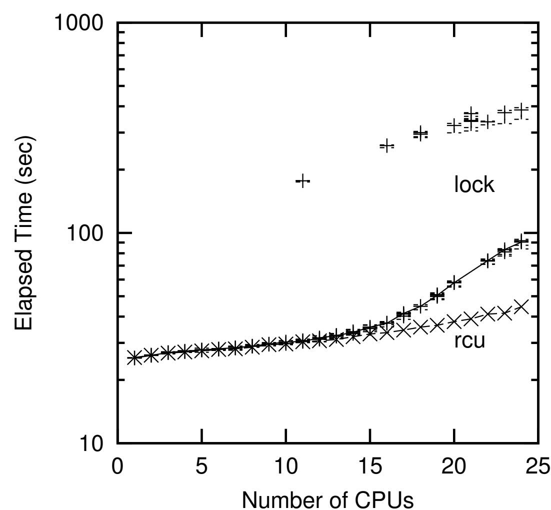


Figure 6.34: PostMark Performance of Hash Table

This lock-free hash table is a drop-in replacement for the many hash tables in the K42 operating system, including the per-file page cache, the PID-to-process mapping table, and the table of simulated page descriptors used by the Linux environment. This hash table is quite valuable in K42, despite the fact that K42 has no global variables. The global hash tables found in many operating systems and applications can be expected to derive even more benefit from this algorithm.

This work demonstrates that NBS may be profitably combined with RCU. Future work

includes exploring additional NBS/RCU combinations, and also eliminating the small amount of locking remaining in this algorithm.

6.8 Other RCU Usage

The following sections briefly overview RCU usage in VM/XA, DYNIX/ptx, K42/Tornado, SuSE Linux, and the Linux 2.6 kernel.

6.8.1 VM/XA

A mechanism resembling RCU [39] is used for per-user-ID tracing in IBM's VM/XA mainframe product [105]. This is an application of the Pure RCU pattern described in Section 5.2.3 on Page 147.

6.8.2 DYNIX/ptx

The DYNIX/ptx development methodology was often extremely qualitative, and multiple improvements were often made simultaneously, as required by competitive pressures and the relatively small size of the DYNIX/ptx development team. There are therefore no accurate measures of the benefits of the RCU implementation for DYNIX/ptx. That said, DYNIX/ptx uses RCU for the following purposes [73]:

1. Distributed lock manager: recovery, lists of callbacks used to report completions and error conditions to user processes, and lists of server and client lock data structures. RCU reduced the complexity of the locking hierarchy, thereby greatly simplifying the deadlock-avoidance code. This subsystem inspired RCU.

2. TCP/IP: routing tables, interface tables, and protocol-control-block lists. This project was used as a testbed to architect DYNIX/ptx's RCU API.

3. Storage-area network (SAN): routing tables and error-injection tables (used for stress testing).

4. Clustered journaling file system: in-core inode lists and distributed-locking data structures.

5. Lock-contention measurement: B* tree used to map from spinlock addresses to the corresponding measurement data (since the spinlocks are only one byte in size, it is not possible to maintain a pointer within each spinlock to the corresponding measurement data).

6. Application regions manager (which is a workload-management subsystem): maintains lists of regions into which processes may be confined.

7. Process management: per-process system-call tables as well as the multi-processor trace data structures used to support user-level debugging of multi-threaded processes.

8. LAN drivers: resolve races between shutting down a LAN device and packets being received by that device. This change applied the Pure RCU pattern.

More information on the implementation and use of RCU in DYNIX/ptx may be found elsewhere [81, 108, 109, 110, 111].

6.8.3 RCU Use in K42

The Tornado and K42 [30] research operating systems independently developed a form of RCU, which is used as follows:

1. To provide existence guarantees throughout these operating systems. These existence guarantees simplify handling of races between use of a data structure and its deletion [10, 30]. The existence guarantees are uses of the RCU Existence Locks pattern described in Section 5.2.4 on Page 151.

2. To identify quiescent states so that implementations of an object can be swapped on the fly while the object is in active use [113]. This is also an example of the Pure RCU pattern described in Section 5.2.3 on Page 147. Note that hot swapping requires that the underlying RCU infrastructure have low grace-period latency.

3. To support a non-blocking hash-table implementation which is used throughout K42, as described in Section 6.7 on Page 221. This implementation uses the RCU

Readers With NBS Writers, Global Version Number, RCU Existence Locks, and Mark Obsolete Objects patterns described in Sections 5.2.6 on Page 158, 5.3.6 on Page 171, 5.2.4 on Page 151, and 5.3.2 on Page 161, respectively.

Use of RCU to provide existence locks is a basic architectural tenet of Tornado and K42, where it reduces lock and memory contention and simplifies locking designs.

6.8.4 RCU Use in Linux

RCU in the Linux 2.4 Kernel

Numerous RCU patches were produced for the Linux 2.4 kernel, as described in Section 4.2 on Page 107. In addition, many of the RCU uses described in Chapter 6 were prototyped in 2.4 kernels.

RCU was also put into production in the 2.4 kernel. SuSE 7.3 Update and later includes an implementation of RCU that is in some ways similar to that in VM/XA. This was the first shipping version of RCU in Linux, and is used to reduce the probability of destructive races that occur in Linux 2.4 kernels during module unloading. This change is an example of the Pure RCU pattern described in Section 5.2.3 on Page 147. Similar code appears in the Linux 2.6.0-test1 kernel.

RCU in the Linux 2.6 Kernel

RCU was introduced incrementally into the Linux 2.5 kernel over some months. This section describes the state of the Linux 2.6.0-test1 kernel.

Linux 2.5.43: Linux 2.5.43 introduced the RCU infrastructure [121], written by Dipankar Sarma. This infrastructure is described in Section 4.4.1 on Page 113. The existing module-unloading code, written by Rusty Russell, made use of this infrastructure. This code is described in Section 6.4 on Page 213.

Linux 2.5.44: Linux 2.5.44 introduced both the `read_barrier_depends()` and the `_rcu` list macros [122], written by Dipankar Sarma. The `_rcu` list macros invoke memory barriers

appropriate for the particular architecture, as suggested by Manfred Spraul. Memory barriers are discussed in detail in Appendix B on Page 322.

Linux 2.5.45: Linux 2.5.45 fixed a bug in RCU's idle-CPU detection [123], written by Dipankar Sarma.

Linux 2.5.46: Linux 2.5.46 introduced an RCU-based implementation of System V IPC [124], written by Mingming Cao. This change is described in detail in Section 6.1 on Page 182.

Linux 2.5.53: Linux 2.5.53 introduced an RCU-based implementation of the IPv4 route cache [125], written by Dipankar Sarma.

Linux 2.5.58: Linux 2.5.58 introduced an RCU-based IPMI (Intelligent Platform Management Interface) driver [126], written by Corey Minyard. This change is similar to the NMI change described in Section 6.3 on Page 211.

Linux 2.5.62: Linux 2.5.62 introduced an RCU-based dcache (directory-entry cache) implementation [127], written by Dipankar Sarma and Maneesh Soni. This change is described in Section 6.2 on Page 195.

Linux 2.5.64: Linux 2.5.64 introduced hlists, including `_rcu` variants, for use in reducing the memory requirements of large hash tables [128], written by Andi Kleen. These hlists are described in Section 4.1 on Page 100.

Linux 2.5.69: Linux 2.5.69 introduced the first installment of RCU as a replacement for brlock (big reader lock) [129], written by Stephen Hemminger.

Linux 2.5.70: Linux 2.5.70 introduced the second installment of RCU as a replacement for brlock [130], again written by Stephen Hemminger. These two installments each contained a large array of changes, which are listed below.

DECNET routing uses RCU to remove brlock. Its approach is very similar to IP route cache, with `__dn_route_output_key()` doing the read-side route-cache lookup.

SNAP protocol registration uses RCU to remove brlock. It applies RCU to a lock-free linked-list search in `find_snap_client()`, in accordance with the Reader-Writer-Lock/RCU Analogy pattern described in Section 5.2.5 on Page 153, and makes use of `synchronize_net()` to defer the `kfree()` of the `datalink_proto()` structure until all searches complete, in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151. The `register_snap_client()` function also makes use of `synchronize_net()` to ensure that, upon return, all subsequent searches will find the new SNAP client. This last example is a use of the Pure RCU pattern described in Section 5.2.3 on Page 147.

802.1Q virtual lan (VLAN) module cleanup uses RCU to remove brlock. It maintains a per-VLAN-group list of device pointers, held in an array indexed by the VLAN ID, also known as the interface index. The array is indexed without locking in accordance with the Reader-Writer-Lock/RCU Analogy pattern described in Section 5.2.5 on Page 153. Removal of entries from this array are followed by a `synchronize_net()` call, which blocks waiting for a grace period to elapse, in accordance with the Pure RCU pattern described in Section 5.2.3 on Page 147. All this aside, the use of global locks to look up the VLAN group indicates that this module would need some work in order to become the basis of a high-performance SMMP VLAN hub. Then again, cost considerations likely force VLAN hubs to be single-CPU devices.

Ethernet bridge uses RCU to remove brlock. It maintains a list of `net_bridge_port` structures, which are subject to lock-free search in `br_get_port_ifindices()`, `br_flood()`, and `br_get_port()` in accordance with the Reader-Writer-Lock/RCU Analogy described in Section 5.2.5 on Page 153. The `del_nbp()` function uses `call_rcu()` to free these structures, in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151.

Combined bridging/routing uses RCU to remove brlock. It maintains a function pointer in `br_should_route_hook()` that is NULL if the module is absent. When the module is unloaded, the pointer is set to NULL, and the unload uses `synchronize_net()`

to block until all tasks that might have seen the non-NULL value have completed their current operation, in accordance with the Pure RCU pattern described in Section 5.2.3 on Page 147.

The packet-handler infrastructure uses RCU to remove brlock. It maintains lists of packet handlers for each packet type, and another list of handlers that is to receive all packet types. These lists are searched in a lock-free manner by `dev_queue_xmit_nit()` and `netif_receive_skb()`, which handle transmitted and received frames, respectively, in accordance with the Reader-Writer-Lock/RCU Analogy pattern described in Section 5.2.5 on Page 153. The `netdev_set_master()`, `unregister_netdevice()`, and `dev_remove_pack()` functions invoke `synchronize_net()` in accordance with RCU Existence Locks described in Section 5.2.4 on Page 151. The packet handlers implement things like Network Interface Tap (NIT), which permits Linux systems to be used as protocol analyzers.

Netfilter uses RCU to remove brlock. There are a number of aspects to netfilter, including netfilter hooks, IP connection tracking, and IP queueing. The `NF_HOOK()` macro and its friends invoke `nf_iterate()` to perform a lock-free traversal of the netfilter hooks, which are registered by `nf_register_hook()` and `nf_unregister_hook()` in accordance with the Reader-Writer-Lock/RCU Analogy described in Section 5.2.5 on Page 153. Both of these latter two functions invoke `synchronize_net()` in order to ensure that the new hook state is seen upon return in accordance with the Pure RCU pattern described in Section 5.2.3 on Page 147. In the case of `nf_unregister_hook()`, the `synchronize_net()` also permits the caller to free the hook data structure immediately upon return, in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151.

Netfilter's IP connection tracking also uses RCU to remove brlock, which was being used to force racing interrupts to complete. There is an `ip_conntrack_lock` that protects data structures, so there may be further opportunities to apply RCU here. RCU is currently used to protect the `helper` list, which is searched by `ip_ct_find_helper()` and `ip_conntrack_alter_reply()` as per the Reader-Writer-Lock/RCU Analogy described in Section 5.2.5 on Page 153. This list is updated by `ip_conntrack_helper_register()` in accordance with the Pure RCU pattern described in Section 5.2.3 on Page 147 and

ip_conntrack_helper_unregister() in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151. A similar situation exists for protocol_list() and for the helpers list used by the network address translation (NAT) facility.

Netfilter's IP queueing uses RCU during module initialization and cleanup to ensure that any racing uses complete before exiting the initialization/cleanup function.

The IPv4 and IPv6 protocol switches use RCU in order to eliminate brlock. The IPv4 inet_protos[] array is protected by RCU. It is searched by icmp_unreach() and ip_local_deliver_finish() in accordance with the Reader-Writer-Lock/RCU Analogy pattern described in Section 5.2.5 on Page 153, and is updated by inet_add_protocol() and inet_del_protocol() in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151. The IPv6 inet6_protos[] array is handled in a similar manner, as are the inetsw and inetsw6.

The IPv4 tunnelling facility uses RCU to remove brlock. The ipip_handler function pointer is protected by RCU, and is updated by the xfrm4_tunnel_register() function and the xfrm4_tunnel_deregister() function in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151. The pointer is dereferenced without locks in ipip_rcv(), whose caller (e.g., ip_local_deliver_finish()) must invoke rcu_read_lock() in accordance with the Reader-Writer-Lock/RCU Analogy pattern described in Section 5.2.5 on Page 153.

The raw socket protocol uses synchronize_net() in the packet_set_ring() function to force interrupts to complete before doing teardown operations in accordance with the RCU Existence Locks pattern described in Section 5.2.4 on Page 151. This use is similar to that of Netfilter's IP connection tracking.

Linux 2.5.71: Linux 2.5.71 introduced some fixes to dcache's d_move() function and to list pointer "poisoning" for RCU-based lists [131], written by Linus Torvalds.

Linux 2.5.73: Linux 2.5.73 introduced RCU-based NMI handling [132], written by John Levon based on a patch by Zwane Mwaikambo. This change is described in more detail in Section 6.3 on Page 211.

6.9 Discussion

This chapter presented several case studies on the application of RCU to operating-system kernels, primarily Linux 2.6.0, and also summarized several tens of uses in VM/XA. DYNIX/ptx, K42/Tornado, and the Linux 2.4 and 2.6 kernels. These uses of RCU typically produced large performance increases, in one case increasing the performance of System V semaphores by more than an order of magnitude, as described in Section 6.1. The complexity of the changes was typically quite modest, in one case requiring only a six-line net addition of code to the Linux 2.6 kernel (thirteen lines added and seven lines deleted), as described in Section 6.5. In some cases, RCU enabled functionality that could be accomplished only with great difficulty using traditional locking schemes, as described in Sections 6.3 and by Gamsa et al. [30] and Appavoo et al. [10]. In other cases, use of RCU allowed other locking primitives to be done away with, for example, brlock was eliminated from the Linux 2.5.69 and 2.5.70 kernels, as discussed in Section 6.8.4.

Each of the case studies used several of the RCU design patterns and transformational patterns described in Chapter 5, validating their use and structure. Future work includes analyzing RCU-related code as it is produced to identify additional RCU patterns and to refine existing RCU patterns.

Chapter 7

Analytical Comparison of RCU and Locking

This chapter presents an analytic comparison of RCU and selected locking primitives. Section 7.1 describes the analytic technique used, Section 7.2 presents derivations valid for low-contention situations, and Section 7.3 discusses the implications of these results. This author adapted the low-contention analysis from earlier publications of his work in this area [68, 81]. The parametric investigation of the RCU/locking breakeven space and the discussion are new work.

7.1 Low-Contention Analytic Methodology

Maintaining low lock contention is essential to attaining high performance in parallel programs. However, as was shown in Section 2.2.7 on Page 25, even programs with negligible lock contention can suffer severe performance degradation due to memory latency and pipeline stalls. Since memory-latency overhead is considerably larger than that of pipeline stalls for many CPUs, as shown in Table 2.1 on Page 32, only memory-latency overhead is considered in this chapter.

The most straightforward way to measure the performance of an algorithm is to simply run it, as was done in Chapters 2 and 6, and as will be done in Chapter 8. However, the direct measurement approach produces results that are specific to a particular workload running on a particular machine. Furthermore, the instrumentation itself may affect the timing and performance measurements.

Alternatively, an algorithm's performance may be evaluated via simulation. The simulator may then be tuned to provide results as a function of memory latency or number of CPUs, but each run still produces results that are specific to a particular workload running on a particular machine, so that exploration of the possible design space is extremely time-consuming.

Traditional design-time methodologies for evaluating the performance of algorithms are based on operation counting [55]. This approach has been refined by many researchers over the decades. Recent work considers the properties of the underlying hardware, weighting the operations by their costs [60]. This hardware-centric approach requires detailed analysis of assembly code, and produces results that are specific to a particular machine.

Although the techniques described in this chapter may be applied to both simulation and analytic operation-counting methodologies, this chapter focuses on analytic methodologies. The scalability design guidelines presented by Unrau et al. [138], namely, preserving parallelism, bounding per-operation overhead, and preserving locality, may profitably be applied at design time when using this technique.

7.1.1 Memory-Latency Model

The approach put forward in this chapter relies on the fact that memory latency is the dominating factor in typical parallel programs. Such programs avoid highly contended locks, leaving the memory latency as the dominating execution cost, since memory accesses are increasingly expensive compared to instruction execution overhead [17, 38, 117].

Since memory latency dominates, we can accurately estimate performance by tracking the flow of data among the CPUs, caches, and memory. For SMMP and NUMA [59] architectures, this data flow is controlled by the cache-coherence protocol, which moves the data in units of cache lines. Figure 7.1 shows a cache line's possible locations relative to a given CPU in a NUMA system. As shown in the figure, a NUMA system is composed of modules called NUMA nodes, which contain both CPUs and memory. Data residing nearer to a given CPU will have shorter access latencies. As the figure shows, data that is already in a given CPU's cache may be accessed with latency t_f. Data located elsewhere on the NUMA node may be accessed with latency t_m, while data located on other NUMA

nodes may be accessed with latency t_s. On large-scale machines where t_s overwhelms t_m and t_f, the latter quantities may often be ignored, further simplifying the analysis, but decreasing accuracy somewhat. If more accuracy is required, the overheads of the individual instructions may be included [60], however, this will usually require that the program be coded and compiled to assembly language, and is often infeasible for large programs.

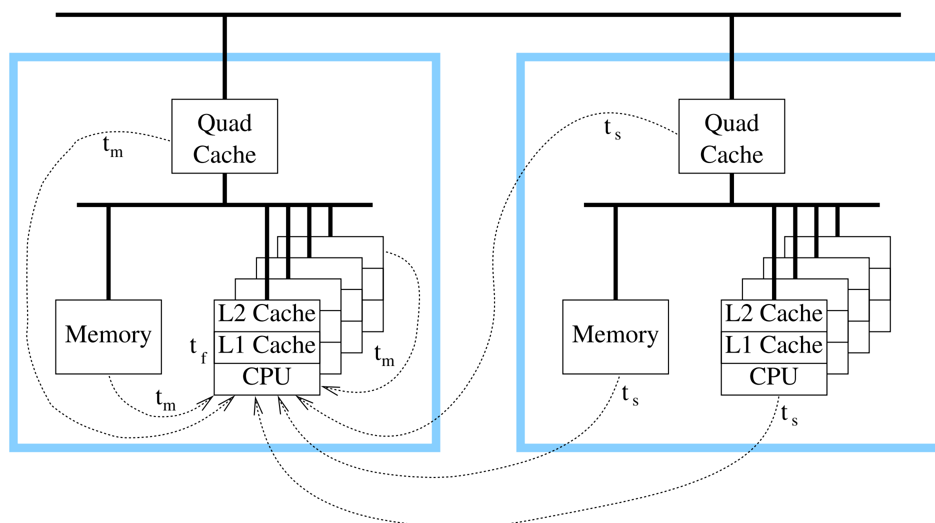


Figure 7.1: NUMA Memory Latency

Once a given data item has been accessed by a CPU, it is cached in that CPU's cache. If the data is located in some other NUMA node's memory, then it will also be cached in the accessing CPU's NUMA node's cache. In both cases, the caching allows subsequent accesses from the same CPU to proceed with much lower latency. Data that has been previously accessed by a given CPU is assumed to reside in that CPU's cache (with access latency t_f). In other words, at low contention, we assume that there is insufficient cache pressure to force data out of a CPU's cache. Most modern CPUs also have a small on-chip cache, which can deliver multiple data items in parallel to the CPU in a single clock. This on-chip cache is modeled as having zero latency, but is assumed only to hold data across a single function.

7.1.2 Conditions and Assumptions

A NUMA system contains n NUMA nodes and m CPUs per NUMA node (two and four, respectively, in the example shown in the figure). The analysis makes the following assumptions:

1. Each NUMA node contains the same number of CPUs.

2. Contention is low and lock-hold times are short compared to the interval between lock acquisitions. This means that the probability of two CPUs attempting to acquire the same lock at the same time is vanishingly small, as is the probability of one CPU attempting to acquire a lock held by another CPU.

3. CPUs acquire locks at random intervals. This means that when a given CPU acquires a lock, that lock was last held, with equal probability, by any of the CPUs. Exclusive and non-exclusive accesses are assumed to occur randomly with probability f and $1 - f$, respectively.

4. The overhead of instructions executed wholly within the microprocessor core is insignificant compared to the overhead of data references that miss the cache. The model can be extended to handle programs with a significant number of "heavyweight" instructions (such as atomic read-modify-write instructions and memory barriers) by adding an additional t_h for these heavyweight instructions. Such extensions are future work.

5. The CPU is assumed to have a single-cycle-access on-chip cache. This cache is considered part of the CPU core, and for purposes of these derivations is called the "on-chip cache". Instruction fetches and stack references (function calls and returns, accesses to auto variables) are assumed to hit this on-chip cache, and are modeled as having zero cost. Indeed, modern microprocessors are frequently able to perform multiple accesses to this on-chip cache in a single clock cycle.

6. Cache pressure is assumed low (outside of the on-chip cache), so that a variable that resides in a given cache remains there until it is invalidated by a write from some

other CPU.

7. Memory-access times are assumed to be independent of the number of copies that appear in different caches. Although directory-based cache-coherence schemes can in theory deviate significantly from this ideal, in practice, this assumption is usually sufficiently accurate [59], particularly for design purposes.

8. Speculative references are ignored. In principle, speculation can result in large quantities of useless but expensive memory references, but in practice, this is often at least partially balanced by the fact that a speculating CPU can fetch multiple data items simultaneously.

7.1.3 Procedural Details

This section gives a step-by-step method of using the latency model to estimate the overhead of an algorithm. It also describes some simplifications that may apply in some commonly occurring situations.

Summary of Nomenclature: Table 7.1 shows the symbols used in the derivations.

Table 7.1: Nomenclature for Lock Cost Derivation

	Definition
f	Fraction of lock acquisitions that require exclusive access to the critical section.
m	Number of CPUs per NUMA node in NUMA systems. Not applicable to SMMP systems. Equations that apply to both SMMP and NUMA systems will define m to be one unless otherwise stated.
n	Number of CPUs (NUMA nodes) in SMMP (NUMA) systems.
r	Ratio of t_s to t_f.
t_c	Time required to access the fine grained hardware clock.
t_s	Time required to complete a "slow" access that misses all local caches.
t_m	Time required to complete a "medium" access that hits memory or a cache shared among a subset of the CPUs. This would be the latency of local memory or of the remote cache in NUMA systems.
t_f	Time required to complete a "fast" access that hits the CPU's cache.

Adaptation to Large-Scale SMMP Machines: The large caches and large memory latencies on large-scale SMMP machines allow them to be modeled in a similar manner. In many cases, substituting t_s for t_m, 1 for n, and n for m reduces a NUMA model to the corresponding SMMP model. Equivalently, one can substitute 0 for t_s, t_s for t_m, 1 for n and n for m.

These substitutions may be used except where the algorithm itself changes form in moving from a NUMA to an SMMP environment. Software that is to run in both NUMA and SMMP environments will generally be coded to operate well in both, often by considering the SMMP system to be a NUMA system with a single large NUMA node.

Use and Simplifications: The model is a four-step process:

1. Analyze the CPU-to-CPU data flow in your algorithm.
2. For each point in the algorithm where a CPU must load a possibly-remote data item, determine the probabilities of that data item being in each of the possible locations relative to the requesting CPU. It is usually best to make a table of the probabilities.
3. For each location, compute the cost.
4. Multiply the probabilities by the corresponding costs, and sum them up to obtain the expected cost.

This process is illustrated on locking primitives in the following sections.

One useful simplification is to set t_f, and possibly t_m, to zero. This greatly simplifies the analysis, and provides accuracy sufficient for many uses, particularly when the ratio r between t_s and t_f is large. The importance of this memory-latency ratio r has long been recognized [136].

A further simplification is to assume that the data is maximally remote each time that a CPU requests it. This further reduces accuracy, but provides a very simple and conservative back-of-the envelope analysis that can easily be applied to large systems during early design.

Note that because the actual behavior depends critically on cache state, actual results can deviate significantly from the analytic results presented in this chapter. For example,

if the CPU cache was fully utilized, the added cache pressure resulting from the larger size of higher-performance locking primitives might well overwhelm their performance benefits. Therefore, analytic results should be used only as guidelines or rules of thumb, and should be double-checked by measuring the actual performance of the running program. Nevertheless, these models have proven quite helpful in practice.

7.2 Low-Contention Derivations

This section presents analytic derivations of the overhead incurred by simple spinlock (Section 7.2.1), distributed reader-writer lock (Section 7.2.2), and RCU (Section 7.2.3). Section 7.2.4 then compares these expressions under various conditions, presenting plots of breakeven loci.

7.2.1 Derivation for Spinlock

A simple spinlock is acquired with a test-and-set instruction sequence. Under low contention, there will be almost no spinning, so the acquisition overhead is just the memory latency to access the cache line containing the lock. This latency is incurred when acquiring and when releasing the lock, and will depend on where the cache line is located, with the different possible locations, probabilities, latencies, and weighted latencies shown in Table 7.2.

The entries in this table are obtained by considering where the lock could have been last held, and, for each possible location, how much it will cost for the current acquisition. In a NUMA system, there are nm CPUs distributed over n NUMA nodes, so there is probability $1/nm$ that the CPU currently acquiring the lock was also the last CPU to acquire it, as shown in the upper-left entry in the table. In this case, the cost to acquire the lock will be just t_f, as shown in the left-most entry of the second row. The weighted latency will be the product of these two quantities, shown in the left-most entry of the third row.

Similarly, there will be probability $(m-1)/nm$ that one of the $m-1$ other CPUs on the same NUMA node as the current CPU last acquired the lock, as shown in the

Table 7.2: Simple Spinlock Access-Type Probabilities and Latencies

	Same CPU	Different CPU, Same Quad	Different Quad
Probability	$1/nm$	$(m-1)/nm$	$(n-1)/n$
Acq. Latency	t_f	t_m	t_s
Wtd. Latency	t_f/nm	$t_m(m-1)/nm$	$t_s(n-1)/n$

upper-middle entry in the table. In this case, the cost to acquire the lock will be t_m, as shown in the middle entry of the second row. Again, the weighted latency will be the product of these two quantities, as shown in the lower-middle entry of the table.

Finally, there will be probability $(n-1)/n$ that one of the CPUs on the other $n-1$ NUMA nodes last acquired the lock, as shown in the upper right entry in the table. In this case, the cost to acquire the lock will be t_s, as shown in the right-hand entry of the middle row. The weighted latency will once again be the product of these two quantities, as shown in the lower right entry of the table.

Under low contention, the overhead of releasing the lock is just the local latency t_f, since there is vanishingly small probability that some other CPU will attempt to acquire the lock while a given CPU holds it. Therefore, the overall NUMA lock-acquisition overhead is obtained by summing the entries in the last row of Table 2 and then adding t_f, as shown in Equation 7.1.

$$\frac{(n-1)\,mt_s + (m-1)\,t_m + (nm+1)\,t_f}{nm} \tag{7.1}$$

An n-CPU SMMP system can be thought of as a single-node NUMA system with n CPUs per NUMA node. The SMMP overhead is therefore obtained by setting n to 1, t_m to t_s, and then m to n, resulting in Equation 7.2.

$$\frac{(n-1)\,t_s + (n+1)\,t_f}{n} \tag{7.2}$$

Both of these expressions approach t_s for large n, validating the common rule of thumb that states that, under low contention, the cost of a spinlock is simply the worst-case memory latency.

Normalizing using the uniform memory-hierarchy model [4] with $t_s = rt_f$ yields the results shown in Equation 7.3 and Equation 7.4.

$$\frac{(n-1)\,mr + (m-1)\,\sqrt{r} + (nm+1)}{nm} \tag{7.3}$$

$$\frac{(n-1)\,r + (n+1)}{n} \tag{7.4}$$

7.2.2 Derivation for Distributed Reader-Writer Spinlock

Distributed reader-writer spinlock is constructed by maintaining a separate simple spinlock per CPU, and an additional simple spinlock to serialize write-side accesses [67]. Each of these locks is in its own cache line in order to prevent false sharing. However, it is possible to interleave multiple distributed reader-writer spinlocks so that the locks for CPU 0 share one cache line, those for CPU 1 a second cache line, and so on. Table 7.3 shows an example layout for a four-CPU system.

Each row in the figure represents a cache line, and each cache line is assumed to hold eight simple spinlocks. Each cache line holds simple spinlocks for one CPU, with the exception of the last cache line, which holds the writer-gate spinlocks. If the entire data structure is thought of as a dense array of forty simple spinlocks, then Lock A occupies indices 0, 8, 16, 24, and 32, Lock B occupies 1, 9, 17, 25, 33, and so on.

Table 7.3: Distributed Reader-Writer Spinlock Memory Layout

	Lock							
CPU	A	B	C	D	E	F	E	F
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
W	32	33	34	35	36	37	38	39

To read-acquire the distributed reader-writer spin-lock, a CPU acquires its lock. If the write fraction f is low, the cost of this acquisition will be roughly t_f. To release a

distributed reader-writer spin-lock, a CPU releases its lock. Again, assuming low f, the cost of the release will be roughly t_f.

To write-acquire the distributed reader-writer spinlock, a CPU first acquires the writer gate, then each of the CPU's spinlocks in order. If the write fraction f is low, the cost of the write-acquisition in this four-CPU example will be roughly $4t_s + t_f$. To release the distributed reader-writer spinlock, a CPU releases the per-CPU locks in order, then the writer gate. Assuming low f, the cost of the release will be roughly $5t_f$. The remainder of this section derives more exact and generally applicable results.

Table 7.4: Unnormalized Probability Matrix for Distributed Reader-Writer Spinlock

	NUMA Node 0				$\cdots$	NUMA Node n			
CPU	0	1	2	3	$\cdots$	$nm-3$	$nm-2$	$nm-1$	nm
Read	$1-f$	$1-f$	$1-f$	$1-f$	$\cdots$	$1-f$	$1-f$	$1-f$	$1-f$
Write	f	f	f	f	$\cdots$	f	f	f	f
Cost	t_f	t_m	t_m	t_m	$\cdots$	t_s	t_s	t_s	t_s

Computing costs for large f is a bit more involved, since a write-side lock will force all CPU's locks into the write-locking CPU's cache, which in turn means that other CPUs' next read acquisitions will cost t_s rather than the usual t_f. Assuming independent interarrival distributions, the probability of a CPU's lock being in its cache is the probability that this CPU did either a read- or write-side acquisition since the last write-side acquisition by any of the other $(n-1)$ CPUs. Similarly, the probability of some other CPU's lock being in a given CPU's cache is the probability that the given CPU did a write-side acquisition since both: (1) the last read-side acquisition by the CPU corresponding to the lock and (2) the last write-side acquisition by one of the $(n-1)$ remaining CPUs. These probabilities may be more easily derived by referring to Table 7.4, which shows the relative frequency and cost of the read- and write-acquisition operations from the viewpoint of CPU 0.

It is important to note that the only events that can affect a given per-CPU lock are read-acquisitions by that CPU (with weighting $1-f$) and write-acquisitions by all CPUs (with weighting nmf), since each read-acquiring CPU affects only its own lock, which each write-acquiring CPU affects all nm CPUs, including itself. Adding the read- and write-acquistion weighting yields a total weighting of $1+(nm-1)f$. This important

quantity will be found in the denominator of many of the subsequent equations.

Read Acquisition and Release

Suppose CPU 0 is read-acquiring the lock. As noted earlier, the only events that can affect the cost are CPU 0's past read-acquisitions and all CPUs' write-acquisitions, for a total weighting of $1 + (nm - 1)f$. Of this, only read- and write-acquisitions, with combined weight of $(1 - f + f) = 1$, will leave CPU 0's element of the distributed reader-writer spinlock in CPU 0's cache.

Therefore, the cost of CPU 0's read operation has probability $1/(1 + (nm - 1)f)$ of being t_f.

Similarly, there is probability $(m-1)f/(1+(nm-1)f)$ that the last operation was a write-acquisition by another CPU on NUMA Node 0, in which case the cost will be t_m.

Finally, there is probability $(nm-m)f/(1+(nm-1)f)$ that the last operation was a write-acquisition by one of the CPUs on the $n-1$ other NUMA nodes, in which case the cost will be t_s.

Weighting these costs by their probability of occurrence gives the expected cost of a read acquisition shown in Equation 7.5.

$$\frac{(nm-m)\,ft_s + (m-1)\,ft_m + t_f}{1 + (nm-1)\,f} \tag{7.5}$$

Read release will impose an additional cost of t_f, as shown in Equation 7.6.

$$\frac{(nm-m)\,ft_s + (m-1)\,ft_m + (2 + (nm-1)\,f)\,t_f}{1 + (nm-1)\,f} \tag{7.6}$$

Write Acquisition

Suppose CPU 0 is write-acquiring the lock. It must first acquire the writer gate, the cost of which was derived in Section 7.2.1. It must then acquire each of the per-CPU locks: its own, those belonging to the other $m-1$ CPUs on the same NUMA node, and those belonging to the $(n-1)m$ CPUs on other NUMA nodes. The cost to acquire its own lock is exactly the same as the read-acquisition case derived previously, resulting in Equation 7.5.

Expressions for the cost of acquiring the other CPUs' locks are derived in the following sections.

Different CPU, Same NUMA Node: If the write-acquiring CPU last write-acquired the lock, the cost will be t_f. If some other CPU on the same NUMA node last acquired the lock for either read or write, the cost will be t_m. Finally, if a CPU on some other NUMA node last write-acquired the lock, the cost will be t_s.

Referring again to Table 7.4, the probability that the write-acquiring CPU last write-acquired the lock is just $f/(1+(nm-1)f)$. The probability that the CPU corresponding to the lock last read- or write-acquired it is $1/(1+(nm-1)f)$, and the probability that another CPU on the same NUMA node last write-acquired the lock is $(m-2)f/(1+(nm-1)f)$, assuming $m \geq 2$. Finally, the probability that a CPU on some other NUMA node last write-acquired the lock is $(nm-m)f/(1+(nm-1)f)$.

Weighting the costs by their respective probabilities of occurrence gives the expected cost of acquiring the per-CPU locks for the CPUs on the same NUMA node as the write-acquiring CPU, as shown in Equation 7.7.

$$\frac{(n-1)\,mft_s + (1+(m-2)\,f)\,t_m + ft_f}{1+(nm-1)\,f} \tag{7.7}$$

Different NUMA Node: If the write-acquiring CPU last write-acquired the lock, the cost will be t_f. If some other CPU on the same NUMA node last write-acquired the lock, the cost will be t_m. Finally, if a CPU on some other NUMA node last write-acquired the lock, or if the owning CPU last read-acquired the lock, the cost will be t_s.

Referring again to Table 7.4, the probability that the write-acquiring CPU last write-acquired the lock is just $f/(1+(nm-1)f)$. The probability that another CPU on the same NUMA node last write-acquired the lock is $(m-1)f/(1+(nm-1)f)$, assuming $m \geq 2$. The probability that the owning CPU last read- or write-acquired the lock is $1/(1+(nm-1)f)$. Finally, the probability that some other CPU on some other NUMA node last write-acquired the lock is $(nm-m-1)f/(1+(nm-1)f$.

Weighting the costs by their respective probabilities of occurrence gives the expected

cost of acquiring the per-CPU locks for the CPUs on different NUMA nodes from the write-acquiring CPU, as shown in Equation 7.8.

$$\frac{(1+(nm-m-1)f)t_s+(m-1)ft_m+ft_f}{1+(nm-1)f} \tag{7.8}$$

Overall Write Acquisition and Release Overhead: The overall write-acquisition and release overhead is the overhead of a simple spinlock (Equations 7.1 and 7.2), plus the overhead of acquiring the per-CPU lock owned by this CPU (Equation 7.5), plus the overhead of acquiring the per-CPU locks owned by the other CPUs on the same NUMA node ($m-1$ times Equation 7.7), plus the overhead of acquiring per-CPU locks owned by the CPUs on other NUMA nodes ($nm-m$ times Equation 7.8), plus the additional overhead of releasing the per-CPU locks (nmt_f). Combining these equations and simplifying yields Equation 7.9.

$$\frac{\left[\begin{array}{c}(n^2m^2+(1-m)nm-m)t_s+\\((m-1)nm+m-1)t_m+\\(n^2m^2+2nm+1)t_f\end{array}\right]}{nm} \tag{7.9}$$

Overall Overhead: The overall overhead is $1-f$ times the overall read overhead (Equation 7.7) plus f times the overall write overhead (Equation 7.9), as shown in Equation 7.10.

$$\frac{\left[\begin{array}{c}\left[\begin{array}{c}\left[\begin{array}{c}n^3m^3-(m+1)n^2m^2+\\(m-1)nm+m\end{array}\right]f^2+\\(2n^2m^2-(2m-1)nm-m)f\end{array}\right]t_s+\\\left[\begin{array}{c}\left[\begin{array}{c}(m-1)n^2m^2-\\(m-1)nm-m+1\end{array}\right]f^2+\\(2(m-1)nm+m-1)f\end{array}\right]t_m+\\\left[\begin{array}{c}(n^3m^3-1)f^2+\\(2n^2m^2-nm+1)f+2nm\end{array}\right]t_f\end{array}\right]}{(nm-1)nmf+nm} \tag{7.10}$$

An n-CPU SMMP system can be thought of as a single-node NUMA system with n CPUs per NUMA node. The SMMP overhead is therefore obtained by setting n to 1, t_s to zero, and then m to n, resulting in Equation 7.11. The same result may be obtained by setting t_m to t_s instead of setting t_s to zero.

$$\frac{\left[\begin{array}{c} \left(\left(n^3 - 2n^2 + 1\right) f^2 + \left(2n^2 - n - 1\right) f\right) t_s + \\ \left(\left(n^3 - 1\right) f^2 + \left(2n^2 - n + 1\right) f + 2n\right) t_f \end{array}\right]}{\left(n^2 - n\right) f + n} \tag{7.11}$$

This expression approaches nft_s for large n and large memory-latency ratios, validating the rule of thumb often used for distributed reader-writer spinlock.

Normalizing with $t_s = rt_f$, $t_m = \sqrt{r}t_f$, and $t_f = 1$ yields the results shown in Equation 7.12 for NUMA systems and Equation 7.13 for SMP systems.

$$\frac{\left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} n^3m^3 - (m+1)\, n^2m^2 + \\ (m-1)\, nm + m \end{array}\right] f^2 + \\ \left(2n^2m^2 - (2m-1)\, nm - m\right) f \end{array}\right] r + \\ \left[\begin{array}{c} \left[\begin{array}{c} (m-1)\, n^2m^2 - \\ (m-1)\, nm - m + 1 \end{array}\right] f^2 + \\ \left(2\,(m-1)\, nm + m - 1\right) f \end{array}\right] \sqrt{r} + \\ \left[\begin{array}{c} \left(n^3m^3 - 1\right) f^2 + \\ \left(2n^2m^2 - nm + 1\right) f + 2nm \end{array}\right] \end{array}\right]}{(nm-1)\, nmf + nm} \tag{7.12}$$

$$\frac{\left[\begin{array}{c} \left(\left(n^3 - 2n^2 + 1\right) f^2 + \left(2n^2 - n - 1\right) f\right) r + \\ \left(\left(n^3 - 1\right) f^2 + \left(2n^2 - n + 1\right) f + 2n\right) \end{array}\right]}{\left(n^2 - n\right) f + n} \tag{7.13}$$

Equation 7.13 is compared to the simple spinlock equivalent (Equation 7.4) in Figure 7.2. The labels in this plot are defined in Table 7.5.

More extensive plots of the costs and breakevens for these and other locking primitives are available in Section 7.2.4 on Page 254 and elsewhere [81].

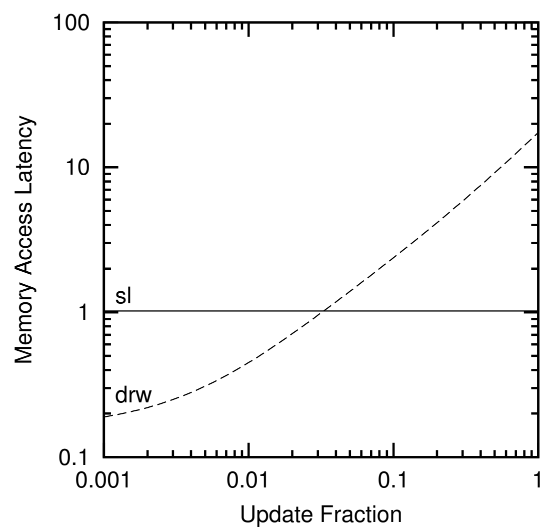


Figure 7.2: Costs of Simple Spinlock and Distributed Reader-Writer Spinlock

Table 7.5: Trace Labels

Label	Description
drw	Distributed (cache-friendly) reader-writer spinlock [67]
sl	Simple spinlock

7.2.3 Derivation for RCU

This derivation of RCU overhead was obtained by applying the methodology described in Section 7.1 to the DYNIX/ptx version 4.4 implementation of RCU. Since the Linux 2.6 kernel uses a similar algorithm, one would expect similar results. In either case, there are four components to RCU overhead:

1. per-scheduling-clock costs. These are incurred on every execution of the per-CPU scheduling-clock interrupt.
2. per-grace-period costs. These are incurred during each RCU grace period.
3. per-batch costs. These are incurred during each RCU "batch", which is the non-zero-sized group of RCU callbacks registered on a given CPU while earlier RCU callbacks on that same CPU were waiting for a grace period to expire. Per-batch costs are incurred only by CPUs that have a batch during a given grace period. These costs are amortized over callbacks making up that batch.
4. per-callback costs. These are incurred for every RCU callback.

Equation 7.14, Equation 7.15, Equation 7.16, and Equation 7.17 give the RCU overhead incurred for each of these four components: per scheduling-clock interrupt, per grace period, per batch, and per callback, respectively:

$$C_h = nmt_c + 3nmt_f \tag{7.14}$$

$$C_g = \begin{bmatrix} (3n + 2nm - m)\, t_s + \\ (2nm + m - 1)\, t_m + \\ (7nm + 1)\, t_f \end{bmatrix} \tag{7.15}$$

$$C_b = 3t_s \tag{7.16}$$

$$C_c = 7t_f \tag{7.17}$$

Again, these were derived by applying the methodology described in Section 7.1 to the RCU implementation in the DYNIX/ptx 4.4 operating-system kernel.

The best-case incremental cost of an RCU callback, given that at least one other callback is a member of the same batch, is just C_c, or $7t_f$.

The worst-case cost of an isolated callback is m times the per-scheduling-clock cost plus the sum of the rest of the costs, as shown in Equation 7.18:

$$C_{wc} = \begin{bmatrix} (3n + 2nm - m + 3)\ t_s + \\ (2nm + m - 1)\ t_m + \\ (3nm^2 + 7nm + 8)\ t_f + \\ nm^2 t_c \end{bmatrix} \tag{7.18}$$

Note that this worst case assumes that at most one CPU per NUMA node passes through its first quiescent state for the current grace period during a given period between scheduling clock invocations. In typical commercial workloads, CPUs will pass through several quiescent states per period.

Typical costs may be computed assuming a system-wide Poisson-distributed inter-arrival rate of λ per grace period, as shown in Equation 7.19, where C_k is the cost of detecting the grace period given that there are k callbacks that have been registered across the system.

$$C_{typ} = \frac{\sum\limits_{k=1}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} C_k}{1 - e^{-\lambda}} \tag{7.19}$$

Here $(\lambda k e^{-\lambda})/k!$ is the Poisson-distributed probability that k callbacks are registered during a given grace period if on average λ of them arrive per grace period. Note that the 0th term of the Poisson distribution is omitted, since there is no RCU overhead if there are no RCU arrivals. The division by $1 - e^{-\lambda}$ corrects for this omission.

Since $e^{-\lambda}$ is independent of k, it may be pulled out from under the summation and combined with the denominator, resulting in the following:

$$C_{typ} = \frac{\sum\limits_{k=1}^{\infty} \frac{\lambda^k}{k!} C_k}{e^{\lambda} - 1} \tag{7.20}$$

The quantity C_k is defined as shown in Equation 7.21. This definition states that we pay the per-scheduling-clock and per-grace-period overhead unconditionally, that we pay the per-batch overhead for each of $N_b(k)$ batches, and that we pay per-callback overhead for each callback.

$$C_k = \frac{C_h + C_g + N_b(k)\, C_b + kC_c}{k} \tag{7.21}$$

The expected number of batches $N_b(k)$ is given by the well-known solution to the occupancy problem shown in Equation 7.22, which assumes that the k callbacks are scattered randomly across the nm CPUs:

$$N_b(k) = nm\left(1 - \left(1 - \frac{1}{nm}\right)^k\right) \tag{7.22}$$

This is just the number of CPUs expected to have batches given nm CPUs and k RCU callbacks.

Substituting Equation 7.22 into Equation 7.21 yields:

$$C_k = \frac{C_h + C_g + nm\left(1 - \left(1 - \frac{1}{nm}\right)^k\right) C_b + kC_c}{k} \tag{7.23}$$

Then substituting Equation 7.23 into Equation 7.20 yields:

$$C_{typ} = \frac{\sum_{k=1}^{\infty} \frac{\lambda^k \left(C_h + C_g + nm\left(1 - \left(1 - \frac{1}{nm}\right)^k\right) C_b + kC_c\right)}{k!k}}{e^\lambda - 1} \tag{7.24}$$

Substituting Equation 7.14, Equation 7.15, Equation 7.16, and Equation 7.17 into Equation 7.24, and then simplifying yields Equation 7.25, which is the desired expression for the typical cost:

$$\frac{f}{e^\lambda - 1} \sum_{k=1}^{\infty} \frac{\lambda^k \begin{bmatrix} (3n + 5nm - m)\, r - \\ 3nm\left(1 - \frac{1}{nm}\right)^k r + \\ (2nm + m - 1)\sqrt{r} + \\ (10nm + 7k + 1) + nmt_c \end{bmatrix}}{k!k} \tag{7.25}$$

Since Equation 7.25 is an infinite sum, the question of convergence naturally arises. To answer this question, first note that the series has the form shown in Equation 7.26:

$$\sum_{k=1}^{\infty} \frac{C_1^k \left(C_2 + C_3^k\right)}{k!k} \tag{7.26}$$

Rearranging yields the following:

$$\sum_{k=1}^{\infty} \left[\frac{C_1^k C_2}{k!k} + \frac{(C_1 C_3)^k}{k!k}\right] \tag{7.27}$$

If the infinite sum of each addend converges, then the infinite sum of the addends taken together must also converge. Therefore, each of the addends may be considered separately.

The constant factor C_1 from first addend may be omitted, since constant factors cannot affect convergence. Then both addends have the same form, and a single proof of convergence applies to both, so that the remaining task is to show that a series of the following form converges:

$$\sum_{k=1}^{\infty} \frac{C^k}{k!k} \tag{7.28}$$

For finite C and k, each term of Equation 7.28 will be finite, and therefore the sum of any finite number of its leading terms will also be finite. Now, choose k greater than $max(2C, 2)$, and note that the ratio of term k to term $k+1$ is given by:

$$\frac{C^k (k+1)!(k+1)}{k!kC^{k+1}} = \frac{(k+1)^2}{kC} \tag{7.29}$$

Because $k > 2C$, this ratio must be greater than two. Therefore, the tail of this series is bounded above by a series of the form:

$$\sum_{k=1}^{\infty} \frac{A}{2^k} \tag{7.30}$$

for sufficiently large A.

This is just a geometric series, which converges. Therefore, the series in Equation 7.25 also converges.

7.2.4 Comparison

This section uses the analytic expressions derived for conditions of low contention to compare simple spinlock, distributed reader-writer lock, and RCU. The following sections show how the comparison is affected by the number of CPUs, by the number of RCU updates that occur per grace period λ, and by the memory-latency ratio r. In all sections not explicitly varying r, the value of r is that for a four-CPU 700MHz Pentium-III.

Varying Number of CPUs

This section explores the relative performance of simple spinlock, distributed reader-writer lock, and RCU for relatively small numbers of CPUs. The results for larger numbers of CPUs may be found elsewhere [81].

Figure 7.3 shows the regions of optimality for very small λ, which is the number of RCU updates per grace period. This is the worst case for RCU because each update must bear the full cost of identifying the grace period. Figures 7.4, 7.5, and 7.6 show the regions of optimality when λ is Poisson-distributed and equal to 0.1, 1.0, and 10.0 updates per grace period, respectively. Figure 7.7 shows the limiting case for large λ. Note that single-CPU realtime testing under Linux 2.6.0 has observed more than 1,600 updates per grace period, so these values for λ are quite conservative. As λ rises, RCU's region of optimality grows, first at the expense of distributed reader-writer lock, then at the expense of simple spinlock. The effect of the number of CPUs is more complex. For small λ (worse for RCU), increasing the number of CPUs shrinks RCU's region of optimality, while for large λ, increasing the number of CPUs actually *increases* RCU's region of optimality. This increase is due to batching. Since all CPUs are running the same workload, adding CPUs increases the probability that multiple RCU updates will share a grace period, thereby amortizing the overhead of grace-period detection. Since neither reader-writer lock nor spinlock can take advantage of any sort of batching, the increased number of CPUs favors RCU for sufficiently large values of λ. For comparison, please note again that values of λ exceeding 1,600 have been observed in single-CPU Linux systems under heavy load. We can therefore expect that, when running real-world workloads, increasing the number of

CPUs will favor RCU.

The sharp corner in the boundary between spinlock's and drw's regions of optimality at $n = 2$ for low λ is an artifact of the integral nature of CPUs. If the plots were expanded to include fractional numbers of CPUs, the boundary would be smooth.

Note that RCU has a significant region of optimality even in single-CPU systems. This is important for user-level programs, since they are typically unable to disable interrupts in order to achieve mutual exclusion. These programs can therefore benefit from RCU even on single-CPU systems.

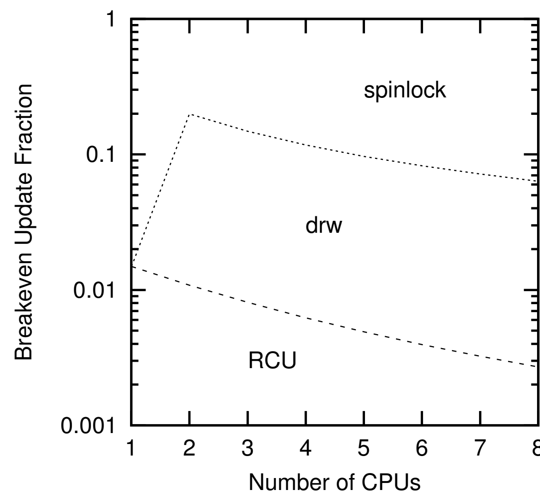


Figure 7.3: Breakevens vs. Number of CPUs, λ Small

Varying Updates per Grace Period

Given the effect that λ has on the regions of optimality, it is interesting to plot the breakeven update fraction f against λ. Figures 7.8, 7.9, and 7.10 present this view for two, four, and eight CPUs, respectively.

Again, as λ increases, RCU's region of optimality again grows. Note that increasing the number of CPUs results in RCU becoming optimal over drw at lower values of λ. It also very slightly decreases RCU's region of optimality for low numbers of CPUs and increases it for high numbers of CPUs, as would be expected given the plots in Section 7.2.4.

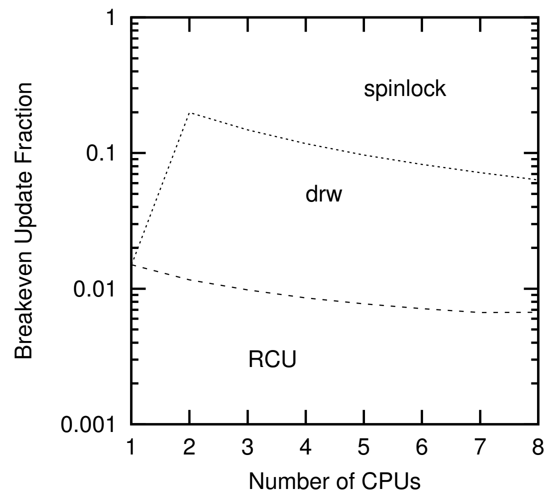


Figure 7.4: Breakevens vs. Number of CPUs, λ=0.1

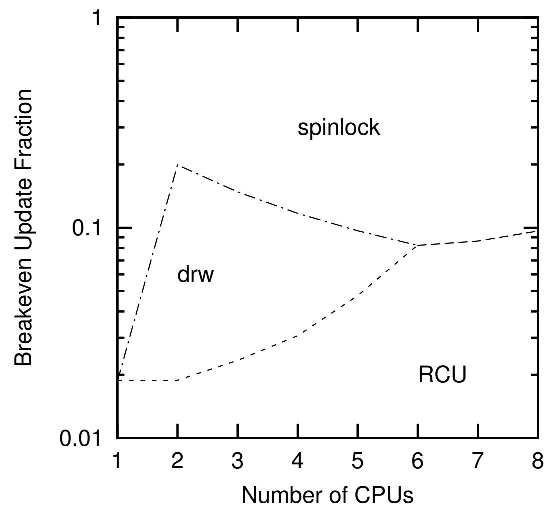


Figure 7.5: Breakevens vs. Number of CPUs, λ=1.0

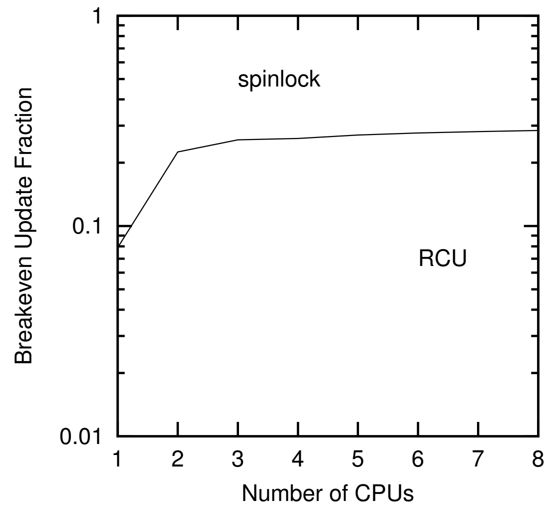


Figure 7.6: Breakevens vs. Number of CPUs, λ=10.0

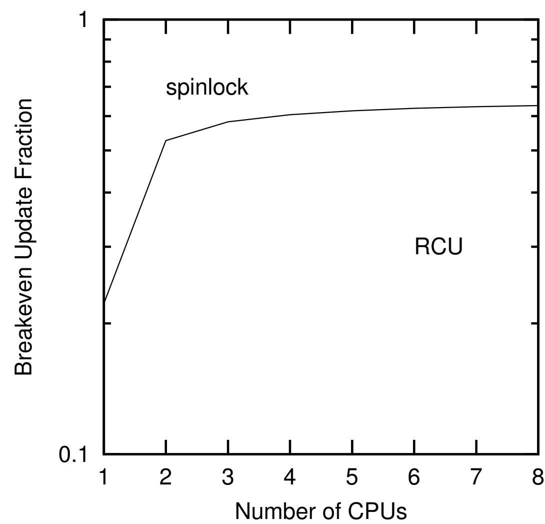


Figure 7.7: Breakevens vs. Number of CPUs, λ Large

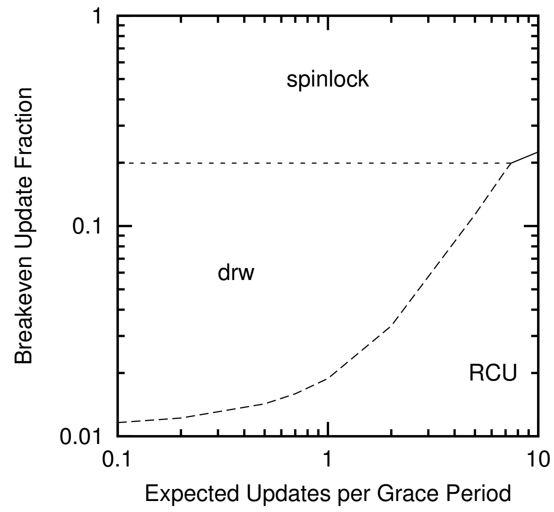


Figure 7.8: Breakevens vs. λ, Two CPUs

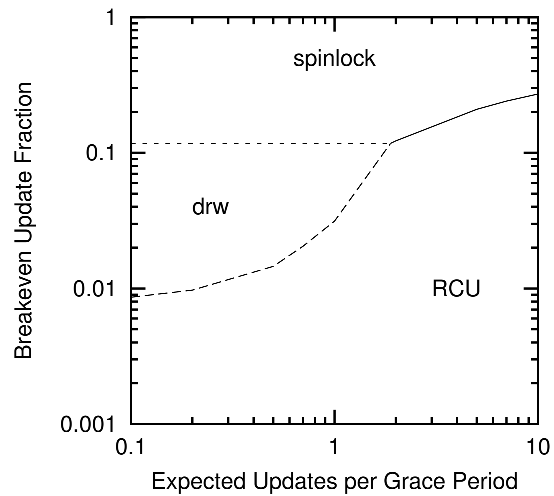


Figure 7.9: Breakevens vs. λ, Four CPUs

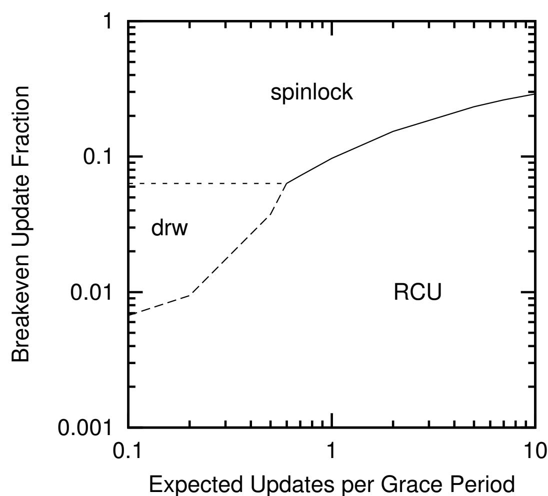


Figure 7.10: Breakevens vs. λ, Eight CPUs

Varying Memory-Latency Ratio

Over the past two decades, the memory-latency ratio has been steadily increasing. This leads one to ask what the effects of continued increase would be. Given the recent appearance of multithreaded CPUs, it is also worth considering the effect of a decrease in the memory-latency ratio.

This is plotted for a two-CPU system for different values of λ in Figures 7.11, 7.12, 7.13, 7.14, and 7.15.

The data is plotted for a four-CPU system for different values of λ in Figures 7.16, 7.17, 7.18, 7.19, and 7.20.

Finally, data is plotted for an eight-CPU system for different values of λ in Figures 7.21, 7.22, 7.23, 7.24, and 7.25.

The effect of varying the memory-latency ratio r depends on the value of the number of updates per grace period λ. For small λ, decreasing r increases the update fraction f for which RCU is optimal. For large λ, the opposite is true–RCU's effectiveness increases with memory-latency ratio. However, for large λ, the breakeven update fraction f is quite large throughout the full range of λ. Therefore, software systems that make heavy use of RCU

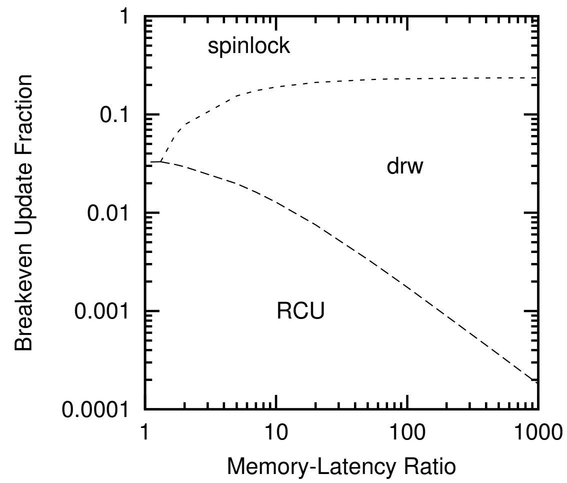


Figure 7.11: Breakevens vs. r, λ Small, Two CPUs

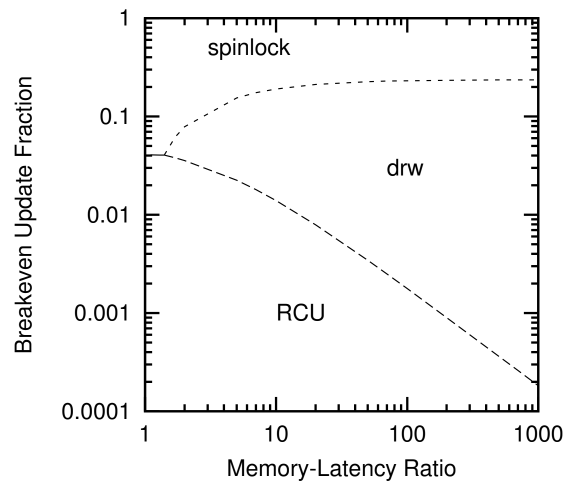


Figure 7.12: Breakevens vs. r, λ=0.1, Two CPUs

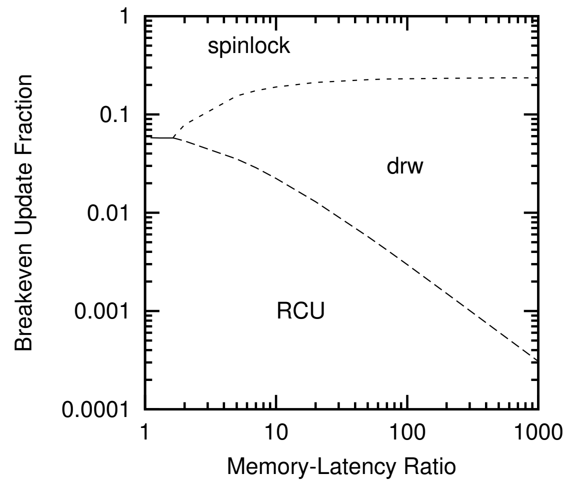


Figure 7.13: Breakevens vs. r, λ=1.0, Two CPUs

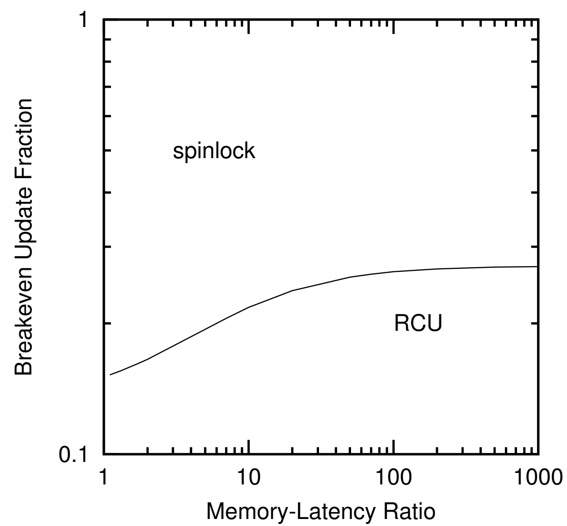


Figure 7.14: Breakevens vs. r, λ=10.0, Two CPUs

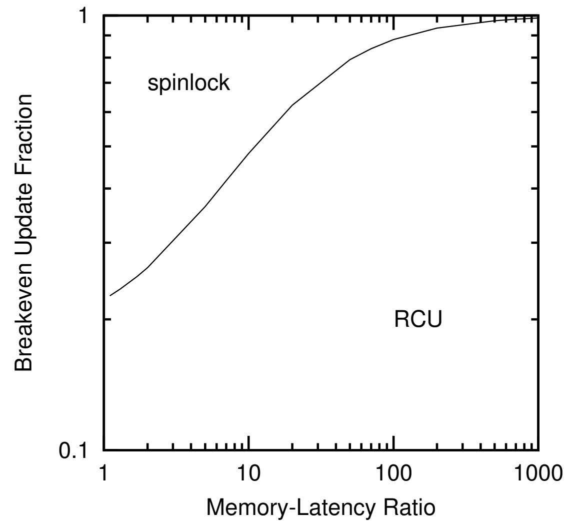


Figure 7.15: Breakevens vs. r, λ Large, Two CPUs

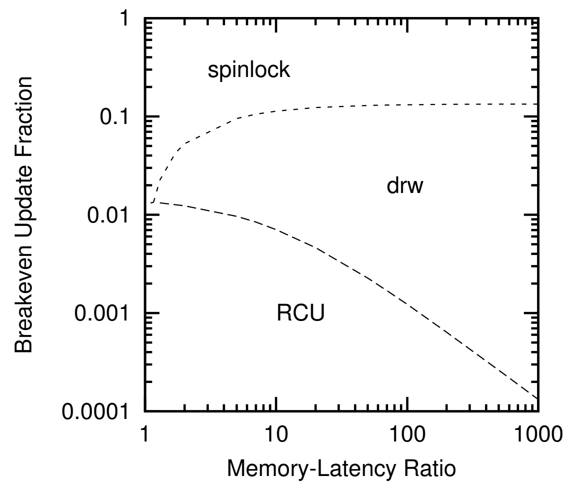


Figure 7.16: Breakevens vs. r, λ Small, Four CPUs

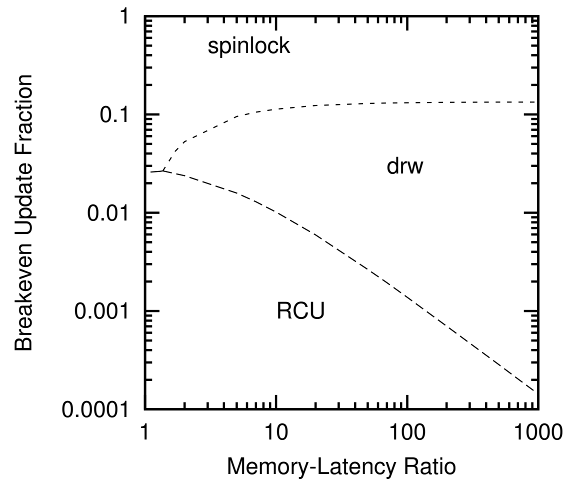


Figure 7.17: Breakevens vs. r, λ=0.1, Four CPUs

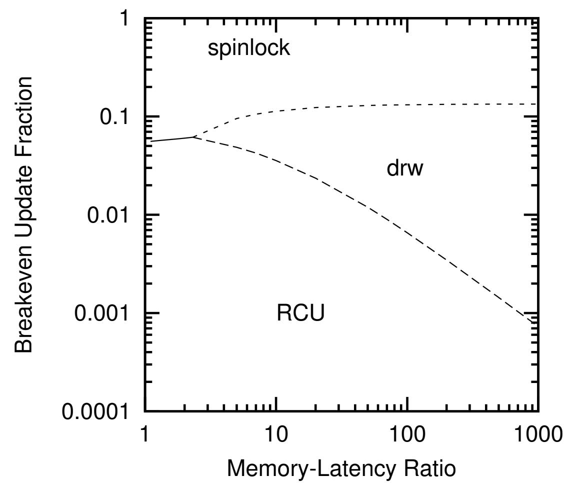


Figure 7.18: Breakevens vs. r, λ=1.0, Four CPUs

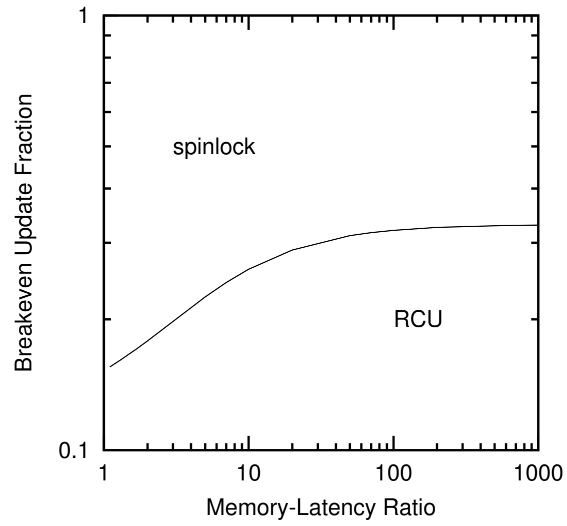


Figure 7.19: Breakevens vs. r, λ=10.0, Four CPUs

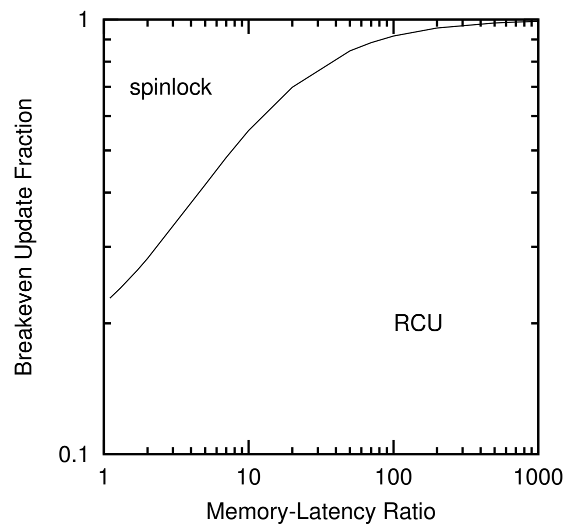


Figure 7.20: Breakevens vs. r, λ Large, Four CPUs

will gain performance benefit from it in over an extremely large range of memory-latency ratios. As noted earlier, the Linux 2.6 kernel running under heavy load experiences values of λ exceeding 1,600, so Linux already falls into the latter category.

This analysis indicates that RCU will have an important role to play regardless of the direction of future memory-latency-ratio trends.

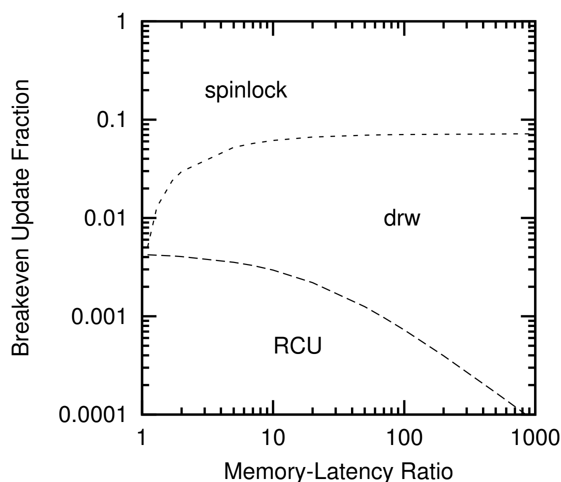


Figure 7.21: Breakevens vs. r, λ Small, Eight CPUs

7.3 Discussion

As shown in the previous sections, the region of optimality for the various locking primitives depends strongly on:

1. the workload, especially as it relates to the expected number of updates per grace period, and

2. the technology, particularly the number of CPUs and the memory-latency ratio.

As can be seen in Figures 7.8 through 7.10, increasing the number of updates per grace period (λ) increases the region of optimality of RCU. One way to accomplish this is to use RCU for algorithms with high update rates, such as the Linux dcache system, which

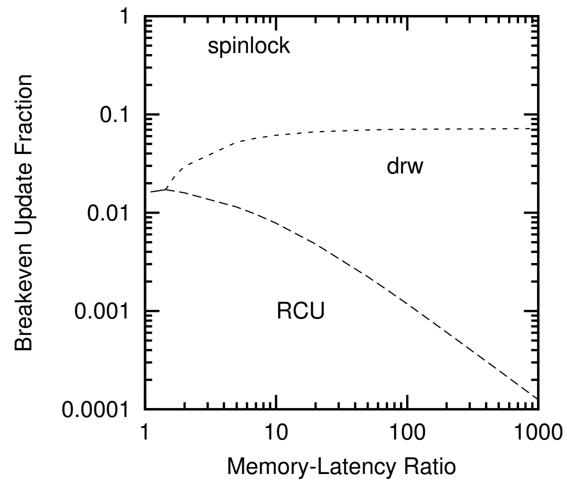


Figure 7.22: Breakevens vs. r, λ=0.1, Eight CPUs

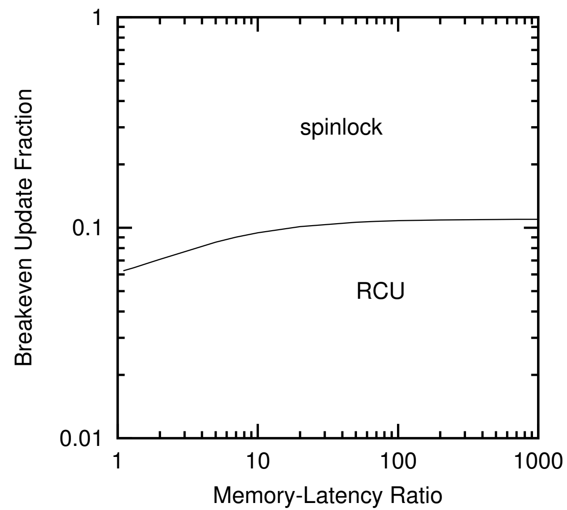


Figure 7.23: Breakevens vs. r, λ=1.0, Eight CPUs

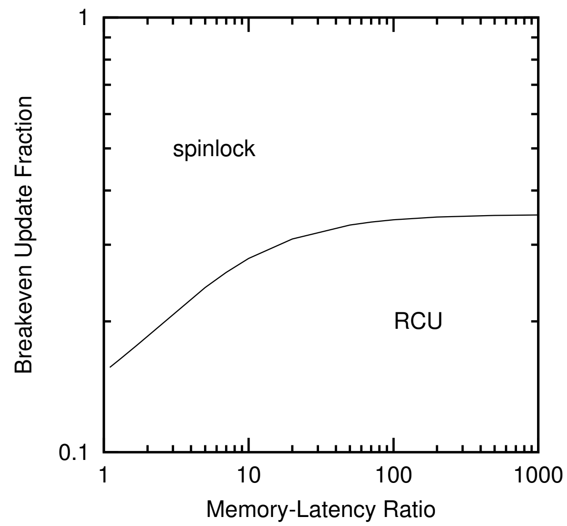


Figure 7.24: Breakevens vs. r, λ=10.0, Eight CPUs

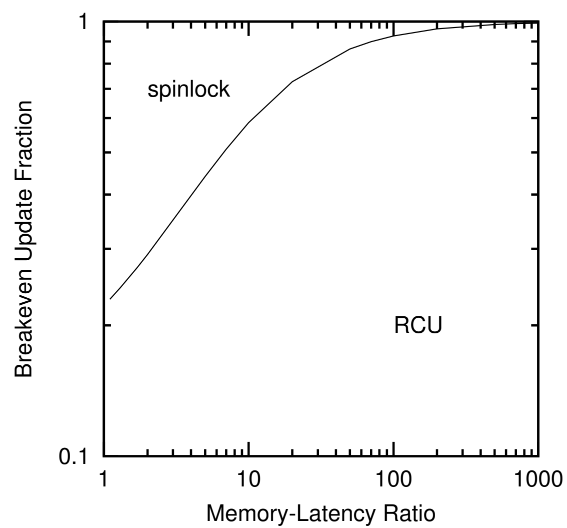


Figure 7.25: Breakevens vs. r, λ Large, Eight CPUs

does an update every time a file is deleted. Future work includes looking at approaches to batch updates and to extend the grace-period duration artificially, the latter spending additional memory to gain CPU efficiency.

This chapter has demonstrated that RCU provides significant performance benefits even on systems with only a single CPU. If the aggregate system update rate λ is low, then RCU's region of optimality decreases with increasing numbers of CPUs, as can be seen in Figures 7.3 and 7.4. In this situation, uses of RCU must then be judged based on the maximum number of CPUs expected during the lifetime of the product, which restricts use of RCU to heavily read-intensive data structures, such as those tracking hardware or software configuration, for example, routing tables. However, increasing λ reverses this situation, as shown in Figures 7.5, 7.6, and 7.7, so that larger numbers of CPUs increase RCU's region of optimality. The Linux 2.6 kernel has been observed running with λ in excess of 1,600, and therefore falls into the latter category.

Chapter 8

Measured Comparison of RCU and Locking

Although the analytic results presented in the previous chapter allow the performance of different synchronization primitives to be compared across a wide variety of hardware platforms and workloads, such results cannot take the place of actual measurements for specific platforms and workloads. To fill this gap, Section 8.1 presents measurements of the hash-table workload described in Section 2.2.4 on Page 21, Section 8.2 compares these measurements to the analytic results derived in Chapter 7, and Section 8.3 presents measurements comparing the different implementations of the RCU infrastructure presented in Chapter 4 on Page 99.

8.1 Comparison to Locking

Given the long history of and deep familiarity with locking, one would expect a new synchronization mechanism such as RCU to be adopted only if it provides some overwhelming advantages over locking. From the discussion in the preceding chapters, one would expect RCU to have overwhelming performance advantages on read-mostly workloads, but that these advantages would wane rapidly with increasing update intensity. One would also expect better RCU update performance with longer grace periods, since the overhead of detecting the grace period could then be amortized over a greater number of updates.

This section tests these expectations, using the same hash-table mini-benchmark that

was presented in Section 2.2.4 on Page 21, running on a 4-CPU 700MHz P-III system.[1] These tests varied the update fraction f and measured the throughput of hash operations (searches and updates) per microsecond for each value of f and for each of the following types of synchronization mechanisms:

ideal: the single-threaded, synchronization-free performance, multiplied by the desired number of CPUs (four in this case).

rcu: the RCU primitive described in this dissertation, using per-hash-chain spinlocks to guard updates.

brlock: the Linux 2.4 kernel's "big reader lock", which provides a lock per CPU, so that reading CPUs acquire only their own lock and writing CPUs acquire all CPUs' locks.

bkt: a per-bucket spinlock, placed in a cacheline separate from the hash-chain header pointers.

bktcl: a per-bucket spinlock colocated in the same cacheline with the hash-chain header pointers.

bktrw: a per-bucket reader-writer spinlock.

globalrw: a global reader-writer spinlock.

global: a global spinlock.

This author has previously presented measurements for this same benchmark taken on other types of CPUs [72].

The results are displayed in Figures 8.1 through 8.4, varying the number of CPUs and the number of operations per grace period λ as follows:

- Figure 8.1 shows results for two CPUs and λ of 10,
- Figure 8.3 shows results for two CPUs and λ of 100,

[1] A variant of Linux RCU devised by Manfred Spraul [115] has run successfully on SMMP machines with as many as 512 CPUs [116], however, detailed performance measurements are not yet available.

- Figure 8.2 shows results for four CPUs and λ of 10, and
- Figure 8.4 shows results for four CPUs and λ of 100.

As can be seen from Figure 8.3, RCU achieves ideal performance on a read-only workload, but is also the best synchronization mechanism up to an update fraction of about 0.4, despite failing to achieve ideal performance. It uniformly beats brlock by more than a factor of two, and manages 50% of the performance of the best synchronization mechanism on an update-only workload. Note that achieving ideal performance on an update-only workload is outside the scope of this dissertation, which focuses on the read-mostly case. Note also that there are significant variations in performance of several of the locking primitives at high update fractions, as can be seen by the non-smooth traces.

Since λ is a per-CPU measure of the number of operations per grace period, Figure 8.1 averages 20 operations per grace period, Figure 8.2 averages 40 operations per grace period, Figure 8.3 averages 200 operations per grace period, and Figure 8.4 averages 400 operations per grace period. These are conservative by comparison to the more than 1,600 updates per grace period observed on single-CPU systems under heavy load during realtime performance testing. RCU achieves very close to ideal performance on a read-mostly workload for the larger numbers of operations per grace period, as expected. The breakeven update fraction is much smaller for the 20-operation-per-grace-period case than for the other three cases, but there is less difference between the breakevens for these other three cases. This is because there is a point of diminishing returns for any amortization process. Note that the breakeven update fraction is slightly higher in Figure 8.4 than in Figure 8.3. This is because the complexity of detecting a grace period increases with increasing number of CPUs, and, unlike the short-grace-period case, this increased overhead for four CPUs is not offset by the greater amortization, again, due to having reached the point of diminishing returns.

8.2 Comparison to Analytic Results

Figure 8.5 shows that the analytically predicted breakeven between uncontended locking and RCU on two CPUs is at an update fraction of roughly 0.3 for λ of 100. This deviates

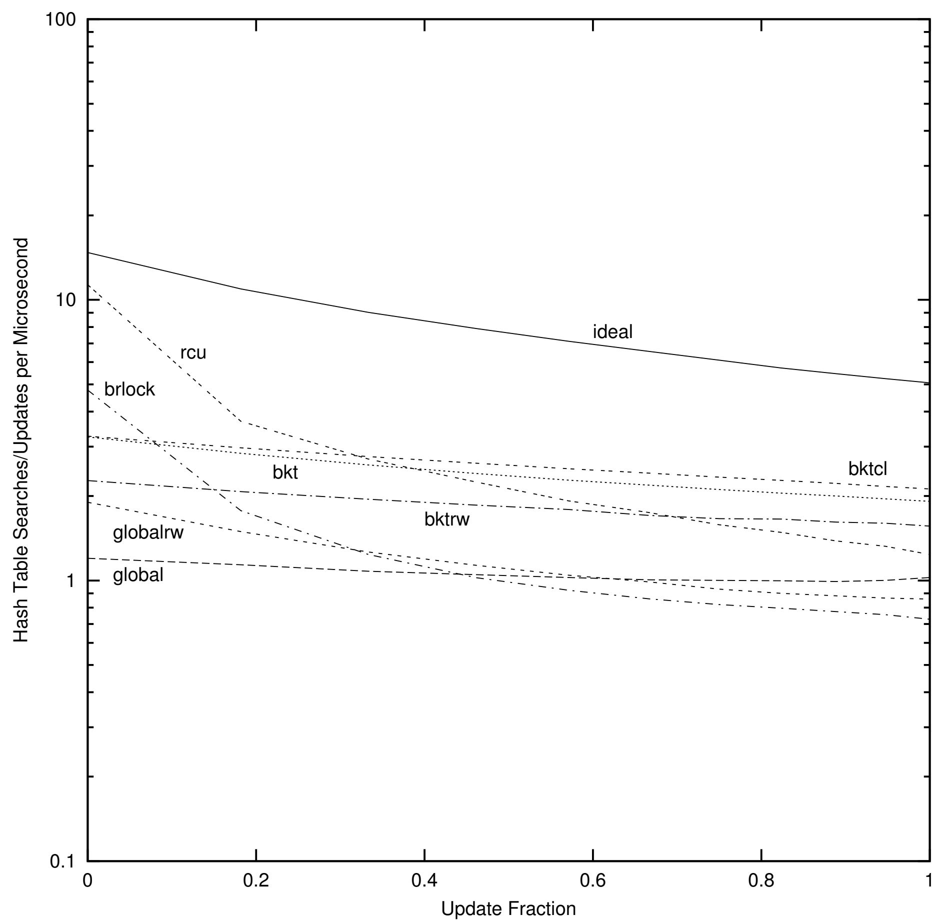


Figure 8.1: Two-CPU Hash Table Performance for Short-Grace-Period Mixed Workload

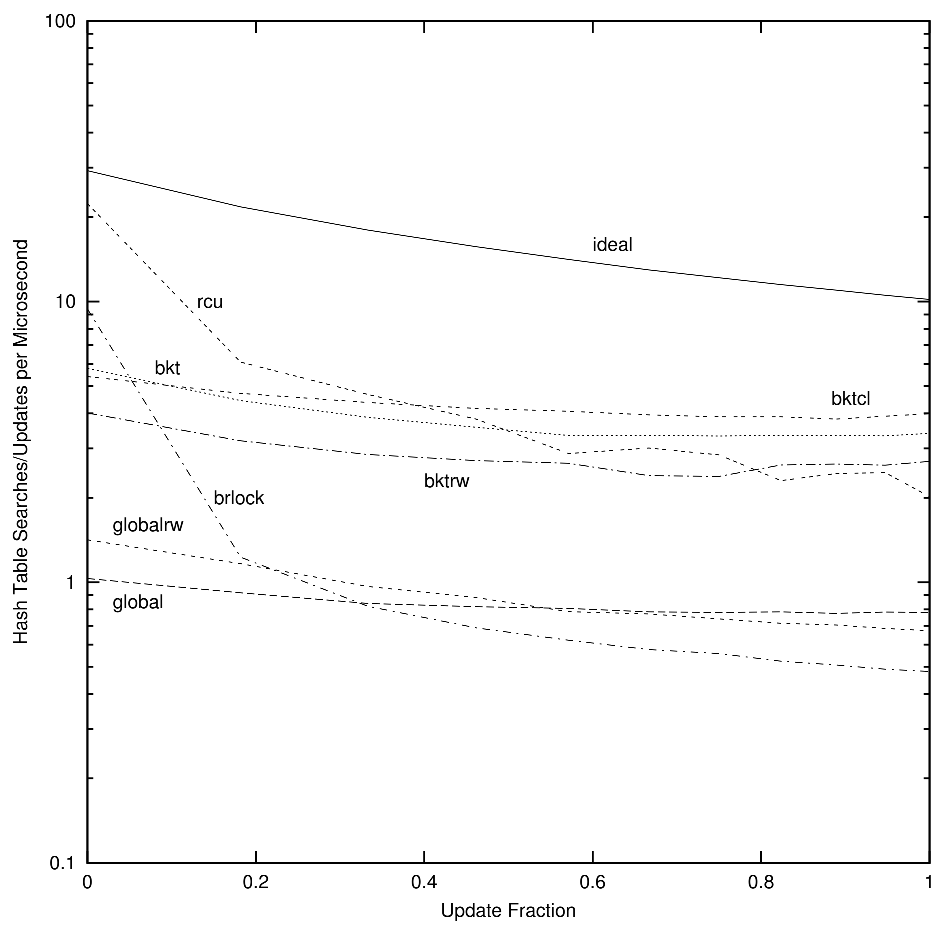


Figure 8.2: Four-CPU Hash Table Performance for Short-Grace-Period Mixed Workload

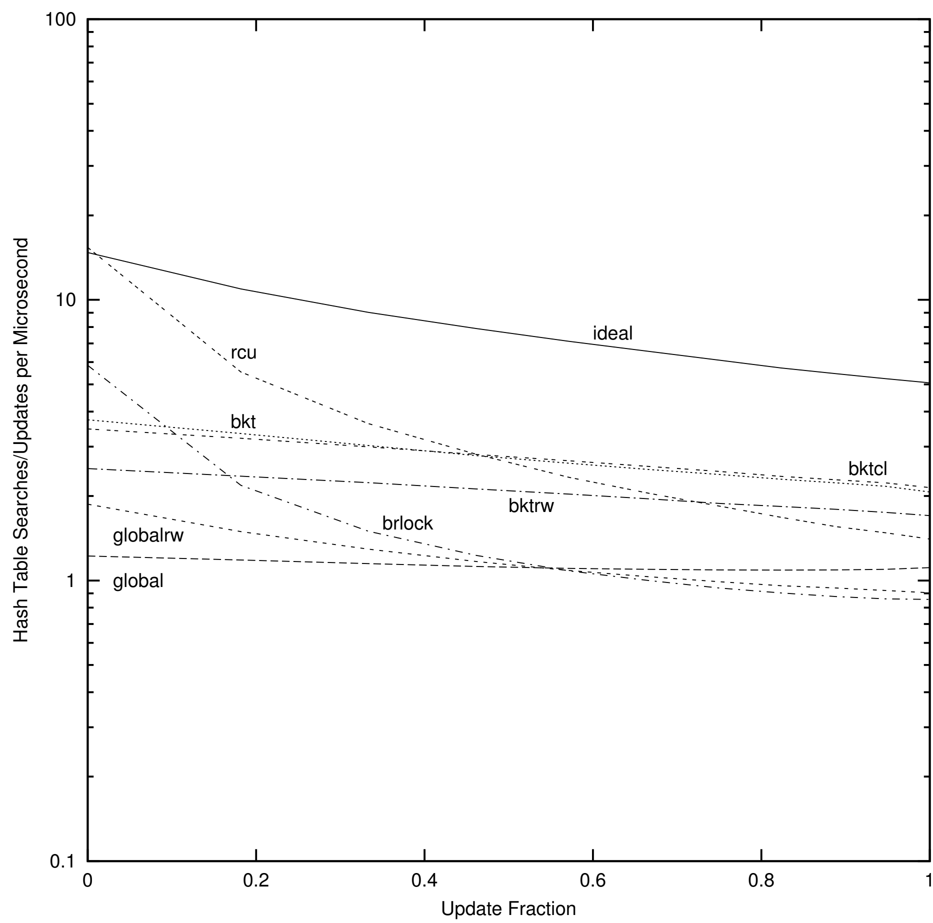


Figure 8.3: Two-CPU Hash Table Performance for Long-Grace-Period Mixed Workload

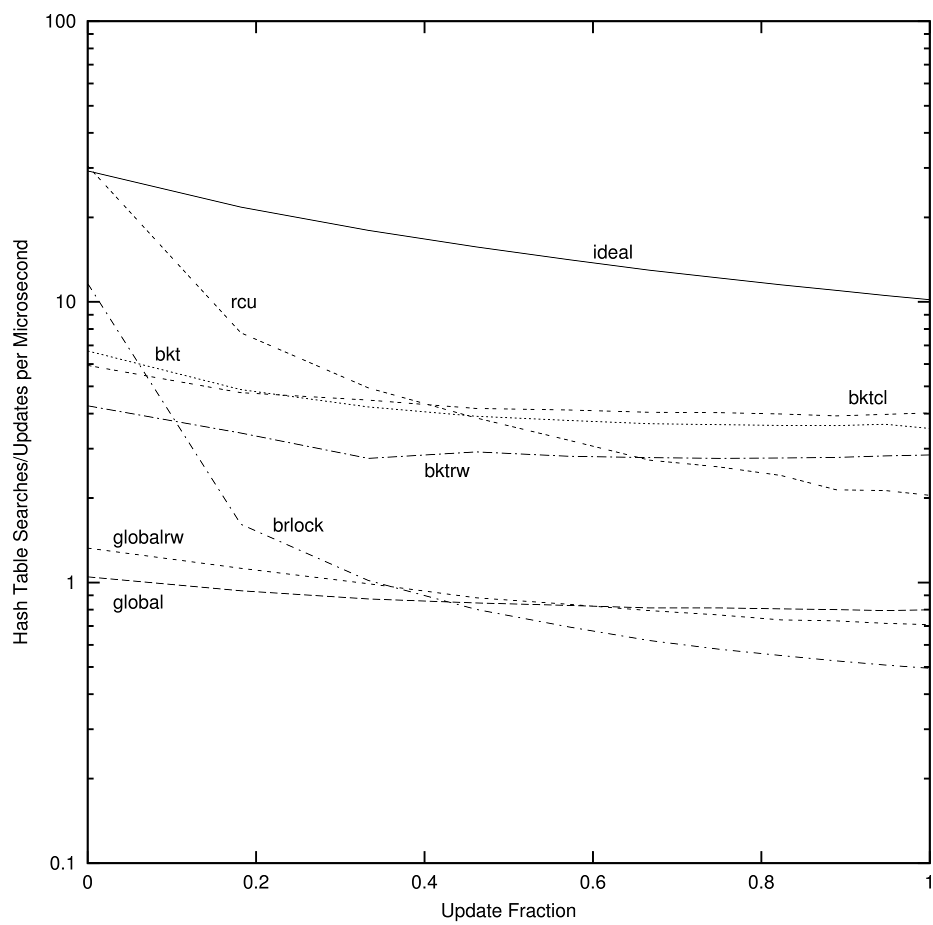


Figure 8.4: Four-CPU Hash Table Performance for Long-Grace-Period Mixed Workload

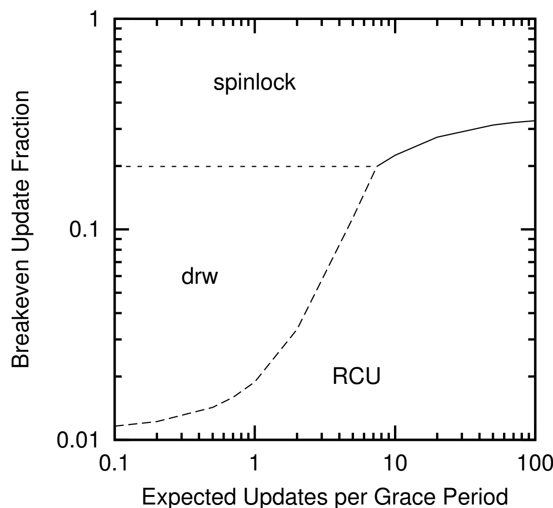


Figure 8.5: RCU Hash Table Breakevens on Two CPUs

significantly from the measured breakeven of about 0.5 for λ of 100 seen in Figure 8.3. However, Figure 8.5 also predicts a breakeven of roughly 0.2 for λ of 10, which is closer to the measured breakeven of about 0.3 for λ of 10 seen in Figure 8.1.

Figure 8.6 shows that the breakeven between uncontended locking and RCU on four CPUs is at an update fraction of roughly 0.3 for λ of 10, and roughly 0.4 for λ of 100. This agrees reasonably well with the measured breakeven of a bit less than 0.4 for λ of 10 seen in Figure 8.2, and with the measured breakeven of a bit more than 0.4 for λ of 100 seen in Figure 8.4.

The author believes that these deviations are due to the following factors:

1. The analysis includes the effects of memory latency, but not of pipeline stalls.

2. The traces for RCU and co-located bucket lock ("bktcl" in the figures) cross at an acute angle, so that any measurement errors are amplified by a large factor in the breakeven.

3. The analysis includes only the overhead of the locking primitives themselves, while the measurements also include the overhead of the hash-table searches and updates.

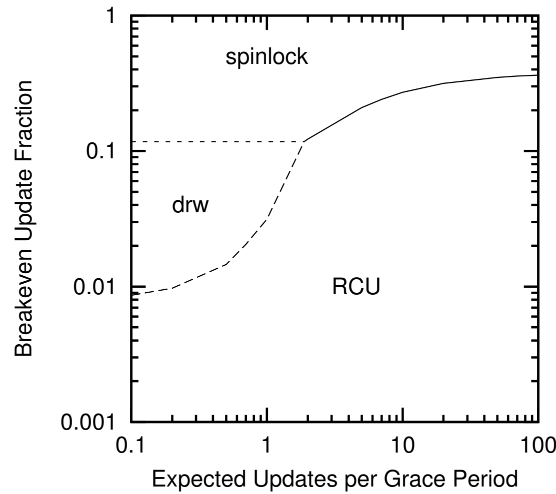


Figure 8.6: RCU Hash Table Breakevens on Four CPUs

4. The analysis was of the DYNIX/ptx 4.4 implementation of RCU, while the measurements were taken from a user-level implementation of RCU running on Linux.

Future work includes more accurately accounting for these factors.

8.3 Evaluation of Techniques for Identifying Grace Periods

Section 8.3.1 compares the alternative Linux algorithms in terms of complexity. The remaining sections discuss performance, with Section 8.3.2 evaluating RCU overhead when there are no RCU-mediated updates in progress, Section 8.3.3 evaluating grace-period latency, and Section 8.3.4 evaluating RCU overhead in the presence of RCU-mediated updates.

8.3.1 RCU Complexity

Table 8.1 shows the number of lines in each algorithm's patch, as was previously reported by myself and others [78]. The "All Archs" column gives the size of the patch applied to all architectures currently in the kernel, while the "One Arch" column gives the size

of each patch applied to only one architecture, namely i386. Architecture-independent patches will have the same number in both columns. The *rcu-taskq* implementation is the simplest, and so might be a good place to start when selecting an RCU implementations.

The *rcu-ltimer* patch works only on the i386 architecture, so the italicized number for "All Archs" is an estimate based on the i386-specific portion of the patch, which simply invokes `RCU_PROCESS_CALLBACKS()` from the `smp_local_timer_interrupt()` function. Later versions of the *rcu-ltimer* patch took advantage of the new architecture-independent `scheduler_tick()` function that was added to the 2.6 kernel. The *rcu-sched* patch contains code to guard against architectures that shut down their CPUs when idle.

Table 8.1: RCU Implementation Complexity

			Size of Unified Diffs	
Name	Section	Page	All Archs	One Arch
rcu-taskq-2.5.3-1.patch	C.1	326	237	237
rcu-poll-2.5.3-1.patch	C.2.3	336	378	378
X-rcu-2.5.3-4.patch	C.2.2	332	424	424
rcu-sched-2.5.3-1.patch	4.4.2	123	575	333
rcu-2.5.3-2.patch	C.2.1	330	603	603
rcu-preempt-2.5.8-3.patch	C.3	343	682	682
rcu-ltimer-2.5.3-1.patch	4.4.1	113	*742*	514

8.3.2 RCU Overhead When Idle

Table 8.2 shows the amount of overhead incurred by each implementation when there is no RCU activity in the system, and was previously published by myself and others [78]. These overheads were determined by examining the source code for each implementation. The *rcu-taskq* implementation does best by this measure, with absolutely no overhead. The *rcu-poll* algorithm is next, with but a single local non-atomic increment in the scheduler. The *rcu-preempt* also incurs overhead on each preemption, as would the others if they were adapted to run in a preemptive kernel.

The *X-rcu*, *rcu-krcud*, and *rcu-ltimer* implementations are quite similar to each other, as was described in Section 4.2 on Page 107. Of these three, *rcu_ltimer* has the lowest

overhead on an idle system.

Table 8.2: RCU Idle Overhead

Name	RCU Idle Memory Refs			
	Switch	Preempt	Timer	Timer Type
X-rcu	1 local		8 local + 1 global + 1 timer	50ms/CPU
rcu	1 local		2 local + 1 global read + 1 global write + 1 timer + #CPU * up()	50ms global
rcu-poll	1 local			
rcu-ltimer	1 local		7 local + 1 global + 1 tasklet	per CPU
rcu-taskq				
rcu-sched	1 global read			
rcu-preempt	1 local	6 local	2 local + 1 global read + 1 global write + 1 timer + #CPU * up()	50ms global

8.3.3 Grace-Period Latency

An important figure of merit for an RCU implementation is the grace period latency. The greater the latency, the more memory is waiting on the internal lists for the current grace period to end. On the other hand, longer latency results in higher efficiency, since the per-callback-batch processing is done less frequently, spreading the overhead over more `call_rcu()` requests. The best tradeoff depends on the workload:

- Systems with very infrequent `call_rcu()` invocations will never gain any performance benefit from batching. Such systems would therefore prefer short grace-period latencies in order to conserve memory.
- Systems with frequent `call_rcu()` invocations would prefer large grace-period latencies in order to amortize the overhead of detecting a grace period over a greater number of `call_rcu()` invocations.
- Systems with extremely frequent `call_rcu()` invocations would again prefer shorter large grace-period latencies, since there is a point of diminishing returns beyond

which increasing the grace-period latency increases memory consumption with negligible reduction in per-callback amortized grace-period-detection overhead.

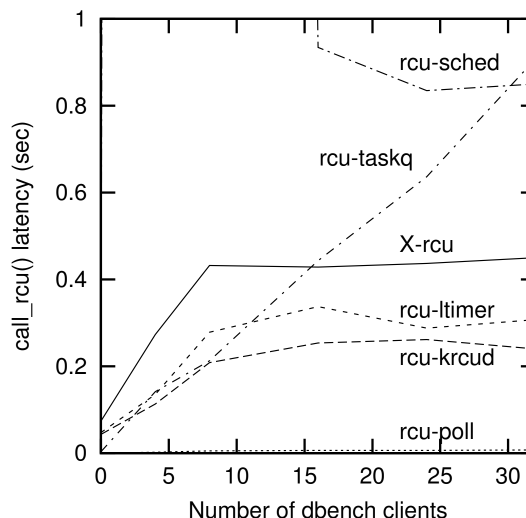


Figure 8.7: call_rcu() Latency Under dbench Load

This latency depends on worst-case kernel codepath length, the workload, and the details of the RCU implementation. Figure 8.7 shows the `call_rcu()` latency for the different RCU algorithms as a function of offered load to the dbench benchmark, which was run by Dipankar Sarma on an 8-CPU 700MHz Xeon system with 1MB L2 caches and 6GB of memory using the dcache-rcu patch on the Linux 2.5.3 kernel, as was published previously by myself and others [78]. The winner by far is *rcu-poll*, which keeps latencies below 10 milliseconds (and below 250 *microseconds* on an idle system) by allowing quiescent states to be detected in parallel and by its aggressive forcing of scheduling when a grace period is required (see Figure 8.8, which shows the same data on a semilog plot). Therefore, *rcu-poll* is suited for systems that either invoke `call_rcu()` infrequently, so that *rcu-poll*'s added overhead is irrelevant, or invoke `call_rcu()` extremely frequently so that significant grace-period-detection amortization is realized despite the very short grace-period latencies. The *X-rcu*, *rcu-ltimer*, and *rcu-krcud* implementations have larger latencies that are well bounded as the number of clients increase. These algorithms are thus suited for

systems that have moderate-to-high rates of `call_rcu()` invocation, because of the performance benefits of batching, amortizing the grace-period detection overhead over greater numbers of callbacks.

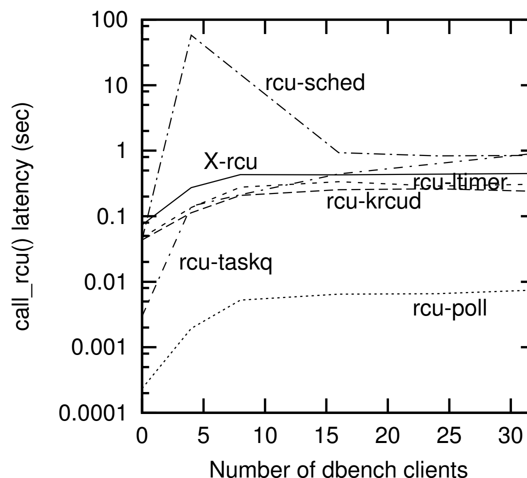


Figure 8.8: call_rcu() Latency Under dbench Load (logscale)

The *rcu-sched* algorithm exhibited very large latencies (14.5 seconds at 8 clients and 57.7 seconds at 4 clients), which are due to early versions of the Linux 2.5 kernel avoiding all scheduler invocations when the number of runnable tasks is exactly equal to the number of CPUs. These extremely large latencies must be greatly reduced before *rcu-sched* may be used on production systems.

The *rcu-taskq* algorithm's latencies increases with increasing numbers of clients, because this algorithm requires the CPUs to pass through quiescent states sequentially, and because keventd (which runs the taskq's) runs at low priority.

Of course, a workload that exercises long-running, non-preemptible code paths in the kernel could greatly lengthen grace-period latency for all of these RCU implementations. However, such long-running code paths have other bad effects, even in absence of RCU, such as grossly degraded response times. Such code paths are therefore bugs that should be fixed.

8.3.4 RCU Overhead When In Use

The performance of RCU-based algorithms critically depends on an efficient implementation of the `call_rcu()` primitive and related RCU infrastructure. The more efficient the implementation, the greater the number of situations that RCU may profitably be applied to. One way to increase the efficiency of the RCU infrastructure is to increase the number of callbacks that are serviced by a single grace period, which can easily be accomplished by arbitrarily extending the duration of each grace period, since any time interval containing a grace period is itself a grace period. However, excessively long grace periods can result in excessive numbers of callbacks pending, which in turn can reduce the overall performance of the system by consuming excessive amounts of memory. In extreme cases, excessive memory consumption can result in system hangs and crashes.

RCU can therefore pose a tradeoff between latency and overhead. This tradeoff is evaluated using two benchmarks, the chat benchmark (a Java-based instant-messaging application) and dbench. These two benchmarks were chosen because they exposed scaling problems in the Linux 2.4 kernel and during early phases of 2.5 development.

The Chat Benchmark

Figure 8.9 compares the performance of the chat benchmark with 20 rooms and 500 messages on a 4-CPU 700MHz Pentium III Xeon system with 1MB L2 caches and 1GB memory. This benchmark was run by Dipankar Sarma using the RCU-based IP-route-cache and FD management patches[2] on the Linux 2.5.3 kernel, and the results were previously published by myself and others [78]. These results show little sensitivity to the RCU algorithm, both compared to base performance and to each other.

Discussion of the chat-benchmark results and of the RCU implementations themselves within the Linux community in mid-2002 resulted in *rcu-taskq* and *X-rcu* being eliminated for performance reasons, *rcu-krcud* for systems administrations and usabilty reasons, and *rcu-preempt* being eliminated due to lack of any prospective users needing preemptible read-side RCU critical sections [11].

[2]The IP-route-cache patch is described by McKenney et al. [78], and the FD management patch is described in Section 6.6 on Page 218.

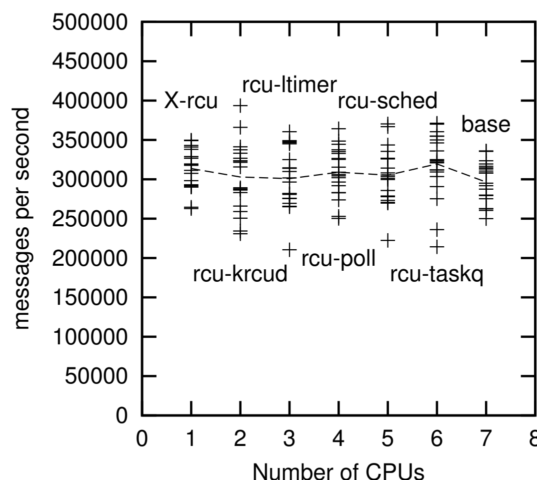


Figure 8.9: RCU Performance on Chat Benchmark

The *rcu-sched* algorithm was retained despite its extreme grace-period latencies because of the possibility that it might be able to process callbacks more efficiently than the two other remaining implementations (*rcu-poll* and *rcu-ltimer*), as was discussed in Section 4.4.2 on Page 123.

The *rcu-ltimer* implementation was modified to take advantage of the new architecture-independent `scheduler_tick()` function, eliminating the architecture-dependent code previously in *rcu-ltimer*.

The *rcu-poll* implementation was modified to use per-CPU callback queues, greatly reducing that implementation's cache thrashing and improving its performance.

The dcachebench Benchmark

The modified *rcu-ltimer* and *rcu-poll* algorithms were then subjected to further performance testing. Table 8.3 compares the performance of *rcu-ltimer* (in the Linux 2.6 kernel), *rcu-sched*, and the parallelized version of *rcu-poll* on the dcachebench directory-cache mini-benchmark [37]. This benchmark was run on a 4-CPU PIII Intel Xeon with 1MB L2 cache and 1GB of memory using the Linux 2.5.46 kernel, and was previously published by myself and others [11].

Table 8.3: dcachebench Comparison

	CPU Utilization		ms/Iteration	
	Avg	Std	Avg	Std
rcu-ltimer	77.52%	0.05%	22.47	0.01
rcu-poll	84.20%	0.13%	22.95	0.03
rcu-sched	81.95%	0.20%	22.46	0.02

These results show that *rcu-ltimer* completes each iteration slightly (but statistically significantly) more quickly than does *rcu-poll*, and with 8.6% less CPU utilization. They also show that `rcu-sched` completes each iteration as fast as does `rcu-ltimer`, but with 5.7% more CPU utilization. Profile results show that *rcu-poll* is incurring significant overhead in the scheduler and in its `force_cpu_reschedule()` function, indicating that, although its cache-thrashing behavior has been addressed, *rcu-poll* is buying its excellent grace-period latency with significantly increased overhead. The *rcu-sched* implementation is unchanged; future work includes optimizing it to eliminate cache thrashing and atomic instructions in order to reduce its overhead.

Therefore, unless grace-period latency is of paramount concern, the *rcu-ltimer* implementation of RCU should be used, and in fact this is the implementation that was accepted into the Linux 2.6 kernel. Should latency become a critical issue in the future, it will be useful to investigate modifications to improve the latency of *rcu-ltimer*.

An optimized *rcu-sched* might beat *rcu-ltimer*'s overhead. If this is the case, reduction of grace-period latency would become a considerably more urgent matter.

8.4 Discussion and Future Scenarios

Observing quiescent states proved more efficient than inducing them, as expected. The most efficient observed-quiescent-state implementation for Linux thus far proved to be the counters-and-barrier (*rcu-ltimer*) implementation described in Section 4.4.1 on Page 113. This greater efficiency was due to use of per-CPU callback queues, which reduces cache-thrashing, and the driving of the barrier computation from the per-CPU-timer interrupt handler, which eliminates context switches from the barrier computation and also reduces

cache thrashing compared to a global implementation such as that of *rcu-poll*, which is described in Section C.2.3 on Page 336.

The analytic predictions of the RCU/locking breakeven update fraction derived in Chapter 7 on Page 235 are reasonably accurate, and show that RCU is optimal for read-mostly workloads with update intensities of up to 30% and, in some cases, even greater. However, the slopes of the cost lines are nearly parallel, which means that increasing RCU grace-period-detection efficiency will allow RCU to be optimal for workloads with significantly greater update fractions. Therefore, further increasing the efficiency of RCU grace-period detection is important future work, since small decreases in RCU overhead will translate into large increases in its region of optimality.

Given the large changes in relative costs of low-level operations shown in Figure 2.1 on Page 20, it is reasonable to ask whether RCU will continue to be useful on future computer systems. Predicting the future course of computer-system evolution has proven to be fraught with peril, so in the following sections The following sections will examine four possible scenarios: (1) Uniprocessor Über Alles, (2) Multithreaded Mania, (3) More of the Same, and (4) Crash Dummies Slamming into the Memory Wall.

8.4.1 Uniprocessor Über Alles

In this scenario, the combination of Moore's-Law increases in CPU clock rate and continued progress in horizontally scaled computing render SMMP systems irrelevant. This scenario is therefore dubbed "Uniprocessor Über Alles", literally, uniprocessors above all else.

These uniprocessor systems would be subject only to instruction overhead, since memory barriers, cache thrashing, and contention do not affect single-CPU systems. In this scenario, RCU is useful only for niche applications, such as interacting with NMIs. It is not clear that an operating system lacking RCU would see the need to adopt it, although operating systems that already implement RCU might continue to do so.

However, recent progress with multithreaded CPUs seems to indicate that this scenario is quite unlikely.

8.4.2 Multithreaded Mania

A less-extreme variant of Uniprocessor Über Alles features uniprocessors with hardware multithreading, and in fact multithreaded CPUs are now standard for many desktop and laptop computer systems. The most aggressively multithreaded CPUs share all levels of cache hierarchy, thereby eliminating CPU-to-CPU memory latency, in turn greatly reducing the performance penalty for traditional synchronization mechanisms. However, a multithreaded CPU would still incur overhead due to contention and to pipeline stalls caused by memory barriers. Furthermore, because all hardware threads share all levels of cache, the cache available to a given hardware thread is a fraction of what it would be on an equivalent single-threaded CPU, which can degrade performance for applications with large cache footprints. There is also some possibility that the restricted amount of cache available will cause RCU-based algorithms to incur performance penalties due to their grace-period-induced additional memory consumption. Investigating this possibility is future work.

However, in order to avoid such performance degradation, a number of multithreaded CPUs and multi-CPU chips partition at least some of the levels of cache on a per-hardware-thread basis. This increases the amount of cache available to each hardware thread, but re-introduces memory latency for cachelines that are passed from one hardware thread to another.

In either case, RCU helps to avoid contention and pipeline-stall overhead due to memory barriers.

8.4.3 More of the Same

The More-of-the-Same scenario assumes that the memory-latency ratios will remain roughly where they are today.

This scenario actually represents a change, since to have more of the same, interconnect performance must begin keeping up with the Moore's-Law increases in core CPU performance. In this scenario, overhead due to pipeline stalls, memory latency, and contention remains significant, and RCU retains the high level of applicability that it enjoys

today.

8.4.4 Crash Dummies Slamming into the Memory Wall

If the memory-latency trends shown in Figure 2.1 on Page 20 continues, then memory latency will continue to grow relative to instruction-execution overhead. Systems such as Linux that have significant use of RCU will find additional use of RCU to be profitable, as shown in Figure 7.20 on Page 264. As can be seen in this figure, if RCU is heavily used, increasing memory-latency ratios give RCU an increasing advantage over other synchronization mechanisms. In contrast, systems with minor use of RCU will require increasingly high degrees of read intensity for use of RCU to pay off, as shown in Figure 7.16 on Page 262. As can be seen in this figure, if RCU is lightly used, increasing memory-latency ratios put RCU at an increasing disadvantage compared to other synchronization mechanisms. Since Linux has been observed with over 1,600 callbacks per grace period under heavy load, it seems safe to say that Linux falls into the former category.

However, if memory latency increases too much relative to instruction-execution overhead, we will likely find ourselves in either the Uniprocessor Über Alles or the Multithreaded Mania scenario, due to the fact that a large body of existing SMMP software would exhibit increasingly poor scalability, causing SMMP systems to in turn exhibit increasingly disadvantageous price-performance measures.

8.4.5 Discussion of Future Scenarios

Low-end systems are already moving in the direction of Multithreaded Mania, as hyperthreading is available even in low-end x86 CPUs, such as those used in desktops and laptops.

Multi-CPU chips are in the offing, and these chips will boast low memory-latency overheads for data fetched out of other CPUs' caches. For these mid-range systems, memory latency is likely to improve as rapidly as instruction-execution rates, so that these systems will fall into the More of the Same scenario.

It is more difficult to predict how high-end systems will progress, since such systems are manufactured in low volumes for a relatively small number of workloads. Historical

trends are consistent with Crash Dummies Slamming into the Memory Wall, but it is conceivable that improved interconnect technology might bring high-end systems into the realm of More of the Same.

The outlook is therefore good for RCU, since it provides substantial performance benefits to all scenarios except for Uniprocessor Über Alles, which seems to be ruled out by the advent of multithreading in low-end systems. The sole exceptions to this are embedded systems where cost and power-consumption constraints still favor single-threaded uniprocessors, and potentially systems with heavily multithreaded CPUs and fully shared cache hierarchies. However, Linux is increasingly the operating system of choice even in the embedded space. Therefore, RCU's future seems quite bright.

Chapter 9

Conclusions and Future Work

This chapter presents a summary and conclusions in Section 9.1, describes directions for future work in Section 9.2, and ends with a parting shot in Section 9.3.

9.1 Summary and Conclusions

This dissertation has:

1. presented performance-related changes in computer-system architecture and outlined the consequent challenges to synchronization mechanisms traditionally used on shared-memory multiprocessor (SMMP) systems;

2. defined a solution, named read-copy update (RCU), to a set of related concurrency problems stemming from these changes and challenges;

3. delineated the relationship between RCU and traditional synchronization mechanisms;

4. using both analytic and empirical means, demonstrated significant performance benefits from use of RCU, ranging from tens of percent to an order of magnitude, both in micro-benchmarks and in system-level formal benchmarks, and with little or no increase in complexity;

5. developed a set of design patterns that permit RCU to be profitably applied to a wide range of synchronization problems; and

6. demonstrated the practical value of RCU by outlining its use in several production systems, two of which have seen extensive datacenter use, and by documenting its acceptance into the Linux 2.6 kernel.

These claims are corroborated in the following sections.

9.1.1 Challenges to Traditional Synchronization Mechanisms

Chapter 1 discussed the problem of synchronization on uniprocessors and SMMP systems. This chapter gave a brief overview of the high costs incurred by traditional synchronization operations and of the scalability limitations they impose on SMMP software systems. These costs include:

1. Instruction-execution overhead.

2. Pipeline-stall overhead.

3. Memory latency.

4. Contention.

Only the first of these overheads has received the full benefit of Moore's-Law increases in performance, but SMMP software must also make heavy use of operations that incur the last three synchronization-related overheads. As a result, SMMP algorithms often perform and scale extremely poorly, as illustrated by a statistical-counter example with order-of-magnitude slowdowns.

The chapter then presented a split-counter algorithm that avoids these costs for increments, but which incurs increased costs for readouts. This split-counter algorithm is an example of an *asymmetric* algorithm that favors increments at the expense of readouts. This design tradeoff is beneficial when increments are more frequent than are readouts, as is the case with many statistical counters. However, the split counter is quite specialized, and therefore does not meet the need for generally applicable synchronization mechanisms.

Since linked data structures are ubiquitous, Chapter 1 then examined the problem of insertion into and deletion from a linked list while permitting synchronization-free readers

to traverse the list. This problem can be solved by using multiple versions of elements and deferring reclamation of versions that have been removed from the list. Versioning with deferred reclamation is a key concept underlying RCU.

9.1.2 Definition of RCU

Chapter 3 on Page 71 provided an overview of RCU, which is a reader-writer synchronization mechanism that takes asymmetric distribution of synchronization overhead to its logical extreme: read-side critical sections incur virtually zero synchronization overhead, containing no locks, no atomic instructions, and, on most architectures, no memory-barrier instructions. RCU therefore achieves near-ideal performance for read-only workloads on most architectures. Write-side critical sections must incur substantial synchronization overhead, deferring destruction and maintaining multiple versions of data structures in order to accommodate the synchronization-free read-side critical sections. In addition, writers must use some synchronization mechanism, such as locking, to ensure orderly updates.

Readers must provide a signal enabling writers to determine when it is safe to complete destructive operations, but this signal may be deferred, permitting a single signal operation to serve multiple read-side RCU critical sections. RCU typically signals writers by non-atomically incrementing a local counter.

These read-side signals are observed by a specialized garbage collector, which carries out destructive operations once all readers have signalled that it is safe to do so. Garbage collectors are typically implemented in a manner similar to a barrier computation, or, on NUMA systems, a combining tree. Production-quality garbage collectors batch destructive operations, so as to amortize signal-observation overhead over many write-side update operations.

Chapter 4 on Page 99 presented the RCU API, along with several types of implementations of the RCU grace-period-detection mechanism. A number of these implementations are in production use in data-center environments.

9.1.3 Relation of RCU to Traditional Synchronization Mechanisms

Chapter 2 on Page 11 discussed related work, starting with synchronization mechanisms used on uniprocessors, then moving to synchronization on SMMP computer systems. The SMMP synchronization mechanisms discussed include exclusive spinlocks, reader-writer spinlocks, partitioning and data locking, asymmetrical reader-writer locking, non-blocking synchronization, transactional memory, exploitation of problem-specific semantics, and mechanisms based on deferred destruction. Table 2.2 on Page 69 summarized the strengths and weaknesses of each approach for read-mostly data structures in operating-system kernels. The deferred-destruction techniques are the most attractive, as they are the only techniques that allow readers to dispense with costly synchronization mechanisms, and in fact RCU is itself an elaboration of these deferred-destruction techniques.

9.1.4 Analytic and Empirical Performance Evaluation

Chapter 6 on Page 179 also presented the performance benefits of RCU, including the order-of-magnitude performance increase enjoyed by the Linux 2.6 kernel's RCU-based System V IPC implementation. This implementation made a very small change to the 2.4 kernel's implementation, requiring the addition of 342 lines of code and the deletion of 191 lines of code, for an net increase of only 151 lines of code on a base of 3,453 lines of code, or less than a 5% increase. This example clearly demonstrates that use of RCU can incur little or no added complexity.

This chapter also described how RCU can be used to provide dynamically changeable non-maskable-interrupt (NMI) handlers. The RCU implementation is quite straightforward. In contrast, any lock-based implementation would be prone to deadlock, since NMIs by definition cannot be masked.

Chapter 7 on Page 235 used analytic techniques to compare the performance of RCU to that of locking, varying memory-latency ratio, number of CPUs, and the read-intensity of the workload. This data shows that RCU works best when heavily used, and, when heavily used, is favored by increasing numbers of CPUs and increasing memory-latency ratios. Chapter 8 on Page 269 then used empirical techniques to compare RCU to locking

on a mini-benchmark, which validated the analytic results. This chapter then compared the performance of a number of grace-period-detection algorithms, finding the *rcu-ltimer* algorithm to be best. As a result of this performance comparison and of the performance work described in Chapter 6, this algorithm was accepted into the Linux 2.6 kernel. This chapter also discussed possible future computer-system performance scenarios, and how RCU would fare in each such scenario. RCU fares quite well in the high-probability scenarios.

9.1.5 Generality of RCU Via Design Patterns

In its raw form, RCU is difficult to use and of limited applicability. Therefore, Chapter 5 on Page 137 presented one set of design patterns that demonstrate how to apply RCU, and another set that transform algorithms into forms that can tolerate the stale and inconsistent data inherent in RCU's use of versioning. Chapter 6 on Page 179 then showed the results of applying these patterns to VM/XA, DYNIX/ptx, K42, SuSE's 2.4 Linux distribution, and the 2.6 Linux kernel.

Four of these design patterns describe how RCU may be used in its raw form:

1. Pure RCU describes how to apply RCU to speed up read-only accesses in cases where stale and inconsistent data may be tolerated. This pattern is especially useful in cases where all outstanding interrupt handlers must complete before an update may be finalized.

2. RCU Existence Locks defer freeing of data-structure elements so that readers may traverse pointers from one element to the next without holding the explicit "existence locks" that would otherwise be required to ensure that the target element was not prematurely freed. RCU Existence Locks can greatly simplify locking designs, since explicit existence locks can be complex and prone to deadlock [30].

3. Reader-Writer-Lock/RCU Analogy describes how to convert an existing reader-writer-lock-based algorithm to use RCU, but only in cases where stale and inconsistent data may be tolerated.

4. RCU Readers With NBS Writers uses non-blocking synchronization rather than locking for updates. The use of RCU simplifies the update code by guaranteeing that deleted elements will not be freed while readers hold references to them.

However, the raw form of RCU exposes readers to stale and inconsistent data, which a large number of algorithms are unable to tolerate. Therefore, the following design patterns may be used to transform such algorithms into forms that are able to tolerate RCU's staleness and inconsistency properties.

1. Mark Obsolete Objects transforms an algorithm that cannot tolerate stale data into one that is able to do so by marking deleted elements. Readers can then ignore any elements that are so marked.

2. Substitute Copy For Original transforms an algorithm that cannot tolerate inconsistent data into one that can, by hiding non-atomic updates behind an atomic substitution operation.

3. Impose Level Of Indirection transforms an algorithm into a form to which Substitute Copy For Original may be applied by grouping related data into one data element which may then be easily substituted.

4. Ordered Update With Ordered Read constrains the ordering of both the update and the read operations so that readers always see consistent data.

5. Global Version Number transforms an algorithm into a form where it can tolerate both stale and inconsistent data by maintaining a global version number and also associating a version number with each element. Readers can then sample the global version number before and after the access, and retry the access if there was an intervening update.

6. Stall Updates prevents excessive update rates from starving readers in the Global Version Number pattern by stalling updates when excessive read-side retries have been executed.

These design patterns have been applied to a wide variety of kernel subsystems ranging from routing tables to directory caches to distributed lock managers. This usage has not been confined to research environments. In fact, as of 1999, seven of the ten largest Oracle database installations ran on the DYNIX/ptx Unix kernel, which made extensive use of RCU.

9.1.6 RCU Has Practical Value

As noted in Section 6.8 on Page 227, this author architected, designed, and implemented the RCU infrastructure used in Sequent's (now IBM's) DYNIX/ptx commercial UNIX operating system. Prior to that, Hennessy, Osisek, and Seigh implemented a mechanism resembling RCU in IBM's VM/XA product for its mainframe systems. Both of these systems have seen heavy use in datacenter environments.

In addition, as noted in Section 6.8.4 on Page 229, an implementation of the RCU grace-period-detection infrastructure, coded by Dipankar Sarma, was accepted into the Linux 2.5.43 kernel, with myself as RCU architect and evangelist. This infrastructure implements the RCU API described in Section 4.1 on Page 100, and has been exploited by the Linux 2.6 kernel's System V IPC, IPv4 route cache, IPMI (Intelligent Platform Management Interface), directory-entry cache, and NMI (non-maskable interrupt) implementations. RCU has also replaced all uses of brlock (big-reader lock) in the Linux 2.6 kernel.

The acceptance of RCU into the Linux kernel has for the first time exposed RCU to a large developer community, and we can therefore expect further significant innovation and change in RCU.

9.1.7 Summary

This dissertation has addressed each of the aims called out at the beginning of this chapter. It has demonstrated that RCU addresses the past few decade's Moore's-Law computer-architecture changes, provides large performance benefits with little or no added complexity, has practical value, and, through the use of appropriate design patterns, is generally applicable in a wide range of operating-system-kernel environments.

9.2 Future Work

Use of RCU has proven quite beneficial in a number of environments. However, it was not until RCU was exposed to a large number of users as a part of this work in Linux that the RCU design patterns were fully refined and codified. This activity has opened up a huge number of research directions, especially in the following areas:

1. RCU infrastructure.
2. RCU design patterns.
3. RCU and non-blocking synchronization.
4. RCU uses in the Linux kernel.
5. Suitability of RCU to other environments.
6. RCU performance.
7. RCU semantics.

The following sections discuss open questions in each of these areas.

9.2.1 RCU Infrastructure

As noted earlier, only with the advent of RCU in Linux has RCU infrastructure been visible to and usable by a a significant number of people. Therefore, it is natural to expect much progress to be made in this area. A few promising areas of investigation follow, but it is expected that additional work will raise additional questions. Specific projects in this area include:

Livelocks and Denial-of-Service Attacks

Since RCU defers deletion using mechanisms based on context-switch monitoring, it is vulnerable to bugs that cause CPUs to loop indefinitely without blocking in the kernel. Such livelock situations can result in indefinite-length grace periods. Of course, livelocks are problematic for other reasons, but it is nevertheless reasonable to ask how RCU might be used to help defend against livelocks and denial-of-service attacks.

1. Evaluate Dipankar Sarma's new `rcu_grace_period()` primitive's ability to aid debugging of livelock situations. This primitive returns how long it has been since this CPU passed through a quiescent state, but only if there is at least one active RCU callback.

2. Create tools that, given a livelocked system, pinpoint the cause of the livelock, possibly based on the `rcu_grace_period()` primitive noted above.

3. Evaluate "bottom-half" variant of RCU developed to handle TCP/IP-based denial-of-service attacks.

Ease of Use

Although the design patterns presented in this dissertation have proven very helpful, it is reasonable to ask what more can be done to make RCU easier to use.

1. Evaluate Dipankar Sarma's new `rcu_barrier()` primitive. This primitive blocks until all outstanding RCU callbacks have completed execution, waiting for all entities waiting on a grace period. It may be useful to the ReiserFS developers in order to better handle unmount of an RCU-exploiting filesystem. The problem here is that outstanding RCU callbacks may need to reference the superblock, so unmount processing must wait for all outstanding RCU callbacks to complete before freeing up the superblock.

   One interesting point: the most obvious implementation of `rcu_barrier()` also waits for all outstanding `rcu_barrier()` invocations. There is therefore no infinite hierarchy of waiting for RCU. In addition, the simplest implementation requires that a given CPU's callbacks be invoked in the same order that they were registered, which has some interesting interactions with CPU hotplug.[1]

2. The explicit read-side memory barriers, such as those required by the DEC Alpha CPU or by the Ordered Updates With Ordered Reads design pattern, are quite

[1]Removal of a CPU from the system requires that any outstanding RCU callbacks be handed off to some other CPU. The most obvious implementation of `rcu_barrier()` requires that the callback lists not be shuffled during such handoffs.

difficult to code, test, and debug. It is therefore reasonable to ask how they might be done away with. One possible avenue is for updates to replace calls to `smp_wmb()` with calls to `synchronize_kernel()`. Since the latter waits for a full grace period, any readers that see updates following the `synchronize_kernel()` are guaranteed to also see the preceding updates. In effect, the `synchronize_kernel()` acts as a "memory-barrier shootdown" that waits until each CPU has executed a memory-barrier instruction. It is also possible to implement a non-blocking memory-barrier-shootdown primitive, as described in Appendix B on Page 322.

It is therefore worthwhile to investigate the complexity and performance implications of the use of such a memory-barrier-shootdown operation. There are a number of ways that such a primitive might be used:

(a) Use memory-barrier shootdown on DEC Alpha only, so that only the current read-side memory barriers can be dispensed with. This would involve replacing write-side `smp_wmb()` calls with `smp_write_barrier_depends()`, which does a memory-barrier shootdown on Alpha and a `smp_wmb()` on other platforms.

(b) Provide memory-barrier shootdown on all architectures so that all read-side memory barriers may be dispensed with. Determine when (if ever) this really works.

(c) Provide three variants of write-side memory barrier:

   i. Current Linux semantics, which matches that of the DEC Alpha wmb (write memory barrier) instruction.

   ii. Semantics of typical CPU's write-side memory barrier, so that read-side memory barriers may be dispensed in situations where there is a data dependency.

   iii. Full memory-barrier shootdown, so that *all* read-side memory barriers may be dispensed with. This seems to require that the update side wait for a grace period between the phases of the update.

Open questions include the performance and complexity consequences of each option,

and which is best in various situations.

Reduced-Overhead Grace-Period Detection

As noted in Section 8.2 on Page 271, small decreases in RCU overhead will result in large increases in its area of applicability. This situation motivates work aimed at reducing the overhead of RCU grace-period-detection overhead.

1. Evaluate *rcu-sched*-based RCU infrastructure, with modifications to avoid the current global counter. This will also require work to handle CPU hotplug, since addition or removal of a CPU must change the token-handoff sequence used by *rcu-sched*.
2. Evaluate implementing memory-barrier shootdown in hardware.
3. Investigate ways of increasing the number of updates per grace period λ, either by batching updates or by artificially extending the grace period. Note that the effective value of λ for Linux is quite large, at least when running heavy filesystem and networking loads. It is therefore likely that Linux will see little or no benefit from this effort.

Hard Realtime and RCU

Under heavy update load, current RCU implementations can execute large numbers of RCU callbacks at the end of a grace period, which in turn can cause realtime scheduling constraints to be missed. This behavior has been observed in the Linux 2.6 kernel. Dipankar Sarma, Andrew Morton, and Robert Love are investigating two approaches to this problem:

1. Enforce a limit on the number of RCU callbacks that can be invoked from the software interrupt (softirq) level. Invoke any excess RCU callbacks from per-CPU kernel daemons, which are preemptible, and therefore do not degrade realtime scheduling latency. Since handing RCU callbacks off to daemons is relatively expensive, only realtime systems would want to take this approach. Interesting questions include how best to identify realtime systems without imposing excess overhead on non-realtime servers or additional configuration problems on system administrators.

2. Where applicable, directly execute the RCU callback from the `call_rcu()` primitive. An interesting open question is whether this can be done automatically, given that `call_rcu()` may be invoked from interrupt handlers or from functions that may be holding a read-side reference to the element being deleted.

9.2.2 RCU Design Patterns

This is the first time that RCU-related design patterns have been published. Although these design patterns will likely continue to be used and that unexpected new design patterns will emerge, there are a number of fundamental questions that remain unanswered:

1. Is there is a complete set of RCU transformational patterns, or, if not, what limitations exist? Here, "complete" means provably able to transform any algorithm into a form that can tolerate both stale and inconsistent data. Given Herlihy's universality results for wait-free synchronization [41], there is reason to hope that there might be such a complete set. However, it is likely that performance would be sacrificed for this generality, for example, by applying the Substitute Copy For Original design pattern to large data structures.

2. If there is a complete set of RCU transformational patterns, under what circumstances does its application result in improved performance and scalability?

3. Are the provisional patterns called out in this dissertation eligible to become full-fledged patterns?

4. Can the set of design patterns presented in this dissertation be refactored into a better form?

9.2.3 RCU and Non-Blocking Synchronization

This is the first time that algorithms combining RCU and non-blocking synchronization have been published. Therefore, again, there are a number of fundamental questions that remain open.

1. RCU can be used in combination with non-blocking synchronization and related techniques in order to remove their dependence on type-safe memory and to remove expensive synchronization operations from the read paths in such techniques. Will such combinations make non-blocking synchronization more broadly applicable?

2. Where does it make sense to combine RCU with non-blocking synchronization and related techniques? One approach to finding the answer would be to combine RCU with a greater variety of non-blocking synchronization algorithms and evaluating the resulting hybrid algorithms.

3. What are the relative merits of combining RCU with non-blocking synchronization?

4. What would be the result of measured and analytic comparisons of various combinations of NBS, RCU, and locking?

5. If the RCU-mediated NBS hash table described in Section 6.7 were extended to use NBS throughout, replacing the use of locking to resolve one of the races, what would the performance and complexity of the resulting implementation be?

9.2.4 RCU Uses in the Linux Kernel

This dissertation analyzes RCU usage in the Linux 2.6.0-test1 kernel. However, there are many more areas within the Linux kernel that are likely to benefit from use of RCU, including the following.

1. Use of RCU in Linux-based filesystems. The ReiserFS developers, particularly Nikita Danilov, have begun this work.

2. Use of RCU in the Linux networking stack. Steve Hemminger has begun this work, as described in Section 6.8.4 on Page 229.

3. Use RCU to replace existing rwlock uses, as appropriate.

4. Upgrade the use of RCU in the directory-entry cache described in Section 6.2 on Page 195 so that pathname walks can be fully lock free.

5. Use of RCU in the Linux virtual memory system. Bill Irwin has begun this work with some patches in his -wli patchtree.

6. Determine whether RCU, or some variant of it, can more completely solve the module-unloading races without imposing heavy overhead or ugly code on modules.

7. Apply the Impose Level Of Indirection (described in Section 5.3.4 on Page 168) to the array-size field in the System V IPC implementation. This would allow the array and its size to be updated atomically, eliminating the need for the explicit read-side memory barriers.

8. Design and implement an FD management modification that uses RCU, but does not degrade the performance of single-threaded processes.

9. Come up with other ways of hiding memory-barrier instructions in various data-structure manipulation primitives, so that RCU may be more easily applied to things other than linked lists.

10. Expanded use of RCU for `tasklist_lock`, perhaps as suggested for `vfs_shared_cred` by Luca Barbieri [12].

11. K42 uses RCU pervasively as a replacement for existence locks. In contrast, Linux has used RCU surgically for specific performance work. It would be interesting to apply RCU pervasively within a particular Linux subsystem, measuring the effect on complexity.

12. Develop an efficient and easy-to-use mechanism and API for reliably reference-counting RCU-protected data structures. Ravikiran Thirumalai is working towards this goal.

13. Most of the uses of RCU in the Linux 2.5 kernel effort were incremental applications of RCU on existing data structures, given how late RCU was introduced into the kernel. It would be interesting to combine RCU with more pervasive restructuring of data and algorithms to obtain greater simplicity and higher performance.

An interesting open question is the extent to which the common-case code paths through Linux can be made lock-free through use of RCU.

9.2.5 Suitability of RCU to Other Environments

Thus far, RCU has been used only in operating-system kernel environments. As noted some time ago [81], there is reason to believe that RCU is also applicable to many other software environments. A short list of possible areas of investigation follows.

1. Evaluate whether a general-purpose set of RCU APIs can efficiently support all environments within the Linux kernel, including different quiescent states. Current indications are that specialized APIs for interrupt handlers will be required.

2. Evaluate use of RCU in various user-level software environments, such as databases, embedded applications, general-purpose libraries, Message-Passing Interface (MPI), Standard Template Adaptive Parallel Library (STAPL), and Java Virtual Machines (JVMs).

3. Peter Strazdins used RCU on a matrix-manipulation problem, and got 5% performance improvement, but with a large increase in complexity. Therefore, it would be interesting to investigate the applicability of RCU to similar user-level applications to see if greater performance could be attained with less complexity.

4. Investigate the possibility of implementing RCU-based library modules for creating, accessing, and maintaining RCU-protected data structures such as lists, hash tables, and trees. Such modules could greatly ease the use of RCU.

5. Evaluate use of RCU in conjunction with the work-crew model.

6. Evaluate using RCU with hardware transactions. Hardware transactions can be thought of as multi-location versions of compare-and-swap or of LL/SC [118].

9.2.6 RCU Performance

Although there has been some analysis of RCU performance and complexity [81], there are still many fruitful avenues of investigation left unexplored:

1. Although grace-period duration has been measured, as described in Section 8.3.3, such measurements are specific to the particular hardware platform, software stack, and workload in use for any given measurement. An analytic evaluation of grace-period duration may be useful in order to design RCU infrastructure and systems with reduced grace-period latency.

2. This dissertation compared CPU overhead and runtimes of a number of synchronization mechanisms. However, RCU's use of deferred deletion causes it to consume more memory and possibly more cache than do other synchronization mechanisms. It would therefore be extremely useful to evaluate memory usage and cache footprint of RCU compared to other synchronization mechanisms, based either on measurements or on the aforementioned analytic evaluation of grace-period duration. The cache-footprint effects may be especially important to heavily multithreaded CPUs with fully shared cache hierarchies.

3. Evaluate likely usefulness of RCU on future computer systems. This might build on the analysis in Section 7.3, but go into more detail and possibly examine more scenarios.

4. Determine how to evaluate "ideal" performance on multithreaded CPUs that share resources, so that linear scaling is prevented by hardware bottlenecks. One likely approach is to simply run an independent copy of the single-threaded benchmark on each hardware thread.

5. Measure performance of RCU on a greater variety of existing CPUs and hardware platforms.

6. Empirical and analytic comparisons of additional RCU-based algorithms to the corresponding locking- and non-blocking-synchronization-based implementations.

7. The RCU implementation within the Linux kernel exploits the semantics of this environment, namely, the knowledge that tasks will context switch sufficiently frequently. Are there other paradigms that permit increased performance and/or decreased complexity through exploiting semantics of the enclosing software environment?

8. The analytic derivations of the low-contention overhead of various synchronization mechanisms presented in Chapter 7 do not include the overhead of pipeline stalls. Although the resulting expressions agreed reasonably well with measurements, it would be interesting to add the effects of pipeline stalls to those analytic expressions.

9. It would be interesting to do an analytic comparison of the various Linux RCU infrastructure implementations.

10. Although the analytic comparisons under low-overhead conditions are informative, it would be interesting to analytically compare various synchronization primitives under conditions of high contention, particularly under conditions of high read-side contention.

11. The analytic expressions derived in Chapter 7 are strictly for the synchronization mechanisms themselves; they do not include the overhead of accessing and updating the protected data structure. To fill this gap, analytic expressions for the overhead of searching and updating a hash table (and other simple data structures) should be created. These expressions should then be combined with the corresponding expressions for the synchronization primitives, and expressions for the resulting breakevens should be derived.

12. Chapter 8 noted a number of sources of error in the measured breakevens between RCU and locking, particularly that the traces for the corresponding overhead curves intersect at an acute angle. It would therefore be valuable to formulate an experimental methodology for more accurately determining the breakevens for RCU and per-bucket locking from measured data. The challenge here is that small errors in the measured overhead translate into large errors in the corresponding breakevens.

13. It may be beneficial to allocate memory for new versions in a cache-friendly manner. For non-preemptive systems, having all versions of a given data element collide in the cache could reduce cache pollution, while for heavily preemptive systems, it might be best to avoid such collisions.

9.2.7 RCU Semantics

This author knows of no work that analyzes RCU semantics. A few of the topics that need investigation are as follows:

1. Formal expression of RCU semantics, perhaps using state-coloring or reachability methods.

2. RCU correctness proofs, both for the infrastructure and for the design patterns.

3. the connections between RCU and computer-related formalisms, such as category theory, algebraic topology, concurrency theory, process algebra, etc.

4. Reusability of algorithms based on RCU. Many synchronization mechanisms and algorithms seem to have been developed in a bottom-up manner, for example, Edler provides implementations for a number of low-level operating-system-kernel operations [26]. In contrast, uses of RCU have tended to be tailored to a specific real-world situation. Are there any formalisms that can gain the best of both worlds, the performance and simplicity benefits delivered by RCU combined with the reusability promised by other approaches?

It is hoped that a more rigorous understanding of RCU's semantics would lead to tools that could assist developers with static and dynamic analysis, for example:

1. Compiler-generated RCU-based algorithms.

2. Tools to automatically identify RCU-susceptible code

3. Tools to automatically check RCU correctness.

The hope is that such tools would permit "ordinary hackers" to successfully produce robust code that provides excellent performance and scalability.

9.3 Concluding Remarks

Perhaps the most fascinating question is "Why has the uptake of RCU been so slow?" The first RCU-related paper that this author is aware of was published in 1980 [56], but

even as late as 2000, RCU was obscure at best. It is as if RCU were an inviting but well-hidden valley nestled deep among the forbidding peaks of SMP scalability. This valley was glimpsed a few times by researchers clinging to one crag or another, but actually settled by only a very few individuals and teams [30, 39, 81].

Speaking as someone who has spent a significant fraction of his career working this valley, I invite you to join us. There have been a succession of unexpected uses to which RCU can profitable be put, an endless supply of questions and topics in need of exploration, and a wealth of exciting possibilities. Life here is good!

Bibliography

[1] ADAMS, G. R. *Concurrent Programming, Principles, and Practices.* Benjamin Cummins, 1991.

[2] ADVANCED MICRO DEVICES. *AMD x86-64 Architecture Programmer's Manual Volumes 1-5*, 2002.

[3] ALEXANDER, C. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[4] ALPERN, B., CARTER, L., FEIG, E., AND SELKER, T. The uniform memory hierarchy model of computation. *Algorithmica 12*, 2/3 (1994), 72–109.

[5] ANDERSON, J. H., AND MOIR, M. Univeral constructions for large objects. In *WDAG: International Workshop on Distributed Algorithms* (1995), LNCS, Springer-Verlag, pp. 168–182.

[6] ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T. A., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (New York, NY, October 1997), pp. 1–14.

[7] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (January 1990), 6–16.

[8] ANDREWS, G. R. Paradigms for process interaction in distributed programs. *ACM Computing Surveys: 23*, 1 (1991), 49–90.

[9] APPAVOO, J. *Optimizing Systems Software with SMMP Distributed Structures.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, In preparation.

[10] APPAVOO, J., HUI, K., SOULES, C. A. N., WISNIEWSKI, R. W., DA SILVA, D. M., KRIEGER, O., AUSLANDER, M. A., EDELSOHN, D. J., GAMSA, B., GANGER,

G. R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal 42*, 1 (January 2003), 60–76.

[11] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310.

[12] Barbieri, L. Re: [patch] initial support for struct vfs_cred. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=103082050621241&w=2` [Viewed: June 23, 2004], August 2002.

[13] Beck, B., and Kasten, B. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 255–275.

[14] Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Technical Conference* (1994), pp. 87–98.

[15] Bonwick, J., and Adams, J. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference, General Track 2001* (2001), pp. 15–33.

[16] Bugnion, E., Devine, S., and Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997), ACM SIGOPS, pp. 143–156.

[17] Burger, D., Goodman, J. R., and Kagi, A. Memory bandwidth limitations of future microprocessors. In *The 23rd Annual International Symposium on Computer Architecture* (New York, NY, May 1996), pp. 78–89.

[18] Cheriton, D. R., and Duda, K. J. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)* (November 1994), USENIX Association, pp. 179–193.

[19] Compaq Computer Corporation. Shared memory, threads, interprocess communication. Available: `http://www.openvms.compaq.com/wizard/wiz_2637.html` [Viewed: June 23, 2004], August 2001.

[20] Control Data Corporation. *3300 Computer System Reference Manual*, 1970.

[21] Coplien, J. O., and Schmidt, D. C., Eds. *Pattern Languages of Program Design*, vol. 1. Addison Wesley, Reading, Massachusetts, 1995.

[22] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with "readers" and "writers". *Communications of the ACM 14*, 10 (October 1971), 667–668.

[23] Cowan, C., Autrey, T., Krasic, C., Pu, C., and Walpole, J. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems (ICCDS'96)* (Annapolis, MD, May 1996), p. 108.

[24] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM 8*, 9 (Sept 1965), 569.

[25] Dijkstra, E. W. *Programming Languages.* Academic Press, 1968, ch. Cooperating Sequential Processes.

[26] Edler, J. *Practical Structures for Parallel Operating Systems.* PhD thesis, Department of Computer Science, New York University, 1995.

[27] Fraser, K. A. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, 2003.

[28] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[29] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. The need for performance-oriented data organization in multiprocessor software. Private communication from Orran Krieger, 1996.

[30] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100.

[31] Garg, A. Parallel STREAMS: a multi-processor implementation. In *USENIX Conference Proceedings* (Berkeley CA, February 1990), USENIX Association, pp. 163–176.

[32] Gelsinger, P. Intel development forum keynote. Available: `http://www.intel.com/pressroom/archive/speeches/gelsinger20040219.htm` [Viewed: June 23, 2004], February 2004.

[33] Ghrachorloo, K. Memory consistency models for shared-memory multiprocessors. Tech. Rep. CSL-TR-95-685, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, December 1995.

[34] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles* (Charleston, SC, December 1999), ACM SIGOPS, pp. 154–169.

[35] Graunke, G., and Thakkar, S. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer 23*, 6 (June 1990), 60–69.

[36] Greenwald, M., and Cheriton, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.

[37] Hartner, B. Dcachbench microbenchmark. Available: `http://www.ibm.com/developerworks/opensource/linuxperf/dcachebench/dcac%hebench.html` [Viewed June 23, 2004], October 2002.

[38] Hennessy, J. L., and Jouppi, N. P. Computer technology and architecture: An evolving interaction. *IEEE Computer* (September 1991), 18–28.

[39] Hennessy, J. P., Osisek, D. L., and Seigh II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989.

[40] Herlihy, M. A methodology for implementing highly concurrent data structures. *Proceedings of the 2nd ACM SIGPLAN Principles and Practice of Parallel Programming* (March 1990), 197–206.

[41] Herlihy, M. Wait-free synchronization. *ACM TOPLAS 11*, 1 (January 1991), 124–149.

[42] Herlihy, M. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (November 1993), 745–770.

[43] Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing* (October 2002), pp. 339–353.

[44] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)* (Providence, RI, May 2003), The Institute of Electrical and Electronics Engineers, Inc., pp. 73–82.

[45] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)* (July 2003), pp. 92–101.

[46] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. *The 20th Annual International Symposium on Computer Architecture* (May 1993), 289–300.

[47] Hoare, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October 1974), 549–557.

[48] Hsieh, W. C., and Weihl, W. E. Scalable reader-writer locks for parallel systems. Tech. Rep. MIT/LCS/TR-521, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[49] IBM Microelectronics and Motorola. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

[50] Inman, J. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 277–298.

[51] Intel Corporation. *i486 Microprocessor Programmer's Reference Manual*, 1990.

[52] Jackson, B. J., McKenney, P. E., Rajamony, R., and Rockhold, R. L. Scalable interruptible queue locks for shared-memory multiprocessor. Tech. Rep. US Patent 6,473,819, US Patent and Trademark Office, Washington, DC, October 2002.

[53] Jacobson, V. Avoid read-side locking via delayed free. Verbal discussion, September 1993.

[54] John, A. Dynamic vnodes – design and implementation. In *USENIX Winter 1995* (New Orleans, LA, January 1995), USENIX Association, pp. 11–23.

[55] Knuth, D. *The Art of Computer Programming.* Addison-Wesley, 1973.

[56] Kung, H. T., and Lehman, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[57] Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (August 1974), 453–455.

[58] Lim, B.-H., and Agarwal, A. Waiting algorithms for synchronization in large-scale multiprocessors. *Transactions on Computer Systems 11*, 3 (August 1993), 253–294.

[59] Lovett, T., and Clapp, R. A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture* (May 1996), pp. 308–317.

[60] Magnusson, P., Landin, A., and Hagersten, E. Efficient software synchronization on large cache coherent multiprocessors. Tech. Rep. T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[61] Manber, U., and Ladner, R. E. Concurrency control in a dynamic search structure. Tech. Rep. 82-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1982.

[62] Manber, U., and Ladner, R. E. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems 9*, 3 (September 1984), 439–455.

[63] Martinez, J. F., and Torrellas, J. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues, International Symposium on Computer Architecture* (Gothenburg, Sweden, June 2001). Available: `http://iacoma.cs.uiuc.edu/iacoma-papers/wmpi_locks.pdf` [Viewed June 23, 2004].

[64] Martinez, J. F., and Torrellas, J. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, October 2002), pp. 18–29.

[65] Massalin, H., and Pu, C. A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Computer Science Department, Columbia University, October 1991.

[66] McKenney, P. E. *Pattern Languages of Program Design*, vol. 2. Addison-Wesley, June 1996, ch. 31: Selecting Locking Designs for Parallel Programs, pp. 501–531.

[67] McKenney, P. E. Selecting locking primitives for parallel programs. *Communications of the ACM 39*, 10 (October 1996), 75–82.

[68] McKenney, P. E. Practical performance estimation on shared-memory multiprocessors. In *Parallel and Distributed Computing and Systems* (Boston, MA, November 1999), pp. 125–134.

[69] McKenney, P. E. Data dependencies and wmb(). Available: `http://lse.sourceforge.net/locking/wmbdd.html` [Viewed June 23, 2004], October 2001.

[70] McKenney, P. E. RFC: patch to allow lock-free traversal of lists with insertion. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=100259266316456&w=2` [Viewed June 23, 2004], October 2001.

[71] McKenney, P. E. Using RCU in the Linux 2.5 kernel. *Linux Journal 1*, 114 (October 2003), 18–26.

[72] McKenney, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, AU, January 2004). Available: `http://www.linux.org.au/conf/2004/abstracts.html#90` `http://www.rdrop.com/users/paulmck/rclock/lockperf.2004.01.17a.pdf` [Viewed June 23, 2004].

[73] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: `http://www.linuxsymposium.org/2001/abstracts/readcopy.php` `http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf` [Viewed June 23, 2004].

[74] McKenney, P. E., Closson, K. A., and Malige, R. Lingering locks with fairness control for multi-node computer systems. Tech. Rep. US Patent 6,480,918, US Patent and Trademark Office, Washington, DC, November 2002.

[75] McKenney, P. E., and Graunke, G. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems* (Tucson, AZ, February 1992), The Institute of Electrical and Electronics Engineers, Inc., pp. 194–199.

[76] McKenney, P. E., McVoy, L., and Tso, T. Y. Re: latest linus-2.5 bk broken. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=102495429606637&w=2` [Viewed June 23, 2004], June 2002.

[77] McKenney, P. E., and Pulamarsetti, C. Optimized function execution for a multiprocessor computer system. Tech. Rep. US Patent 6,418,517, US Patent and Trademark Office, Washington, DC, July 2002.

[78] McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367. Available: `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz` [Viewed June 23, 2004].

[79] McKenney, P. E., Sarma, D., and Soni, M. Scaling dcache with RCU. *Linux Journal 1*, 118 (January 2004), 38–46.

[80] McKenney, P. E., and Slingwine, J. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings* (Berkeley CA, February 1993), USENIX Association, pp. 295–306.

[81] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[82] McVoy, L. Scaling Linux with (partially) CC clusters. Available: `http://www.bitmover.com/ml/` [Viewed June 23, 2004], August 2001.

[83] McVoy, L. SMP clusters. BOF Session at Ottawa Linux Symposium, June 2002.

[84] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems 9*, 1 (February 1991), 21–65.

[85] Mellor-Crummey, J. M., and Scott, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP* (Williamsburg, VA, April 1991), pp. 106–113.

[86] Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture* (August 2002), pp. 73–82.

[87] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (August 2002), pp. 21–30.

[88] Moore, G. No exponential is forever–but we can delay forever. In *IBM Academy of Technology 2003 Annual Meeting* (San Francisco, CA, October 2003).

[89] Oliver, G. J. OS-3 question. Message-ID: <3FBEB230.6090700@ao.com>, 2003.

[90] Presotto, D., and Winterbottom, P. The organization of networks in Plan 9. In *1993 Winter USENIX* (January 1993), USENIX Association, pp. 271–280.

[91] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Copper Mountain, CO, December 1995), pp. 314–321.

[92] Pu, C., Massalin, H., and Ioannidis, J. The Synthesis kernel. *Computing Systems 1*, 1 (January 1988), 11–32.

[93] Pugh, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[94] Pugh, W. A probabalistic alternative to balanced trees. *Communications of the ACM 33*, 6 (June 1990), 668–676.

[95] Radović, Z., and Hagersten, E. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (Baltimore, Maryland, USA, November 2002), The Institute of Electrical and Electronics Engineers, Inc., pp. 1–13.

[96] Radović, Z., and Hagersten, E. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)* (Anaheim, California, USA, February 2003), pp. 241–252.

[97] Rajwar, R., and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Austin, TX, December 2001), The Institute of Electrical and Electronics Engineers, Inc., pp. 294–305.

[98] Rajwar, R., and Goodman, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (Austin, TX, October 2002), pp. 5–17.

[99] Reiman, M. I., and Wright, P. E. Performance analysis of concurrent-read exclusive-write. *ACM* (February 1991), 168–177.

[100] Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E., and Gupta, A. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, December 1995), ACM SIGOPS, pp. 285–298.

[101] Sarma, D. Some dcache_rcu benchmark numbers. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=103462075416638&w=2` [Viewed June 23, 2004], October 2002.

[102] Sarma, D. specweb99: dcache scalability results. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=102645767914212&w=2` [Viewed June 23, 2004], July 2002.

[103] Satyanarayanan, M. Scalable, secure, and highly available distributed file access. *IEEE Computer 23*, 5 (1990), 9–21.

[104] Saxena, S., Peacock, J. K., Yang, F., Verma, V., and Krishnan, M. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. In *1993 Winter USENIX* (January 1993), USENIX Association, pp. 85–96.

[105] Seigh II, J. W. Read copy update. email correspondence, March 2003.

[106] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.

[107] Sequent Computer Systems, Inc. *DYNIX/ptx STREAMS Kernel Programming Guide*, 1990.

[108] Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,442,758, US Patent and Trademark Office, Washington, DC, August 1995.

[109] Slingwine, J. D., and McKenney, P. E. Method for maintaining data coherency using thread activity summaries in a multicomputer system. Tech. Rep. US Patent 5,608,893, US Patent and Trademark Office, Washington, DC, March 1997.

[110] Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,727,209, US Patent and Trademark Office, Washington, DC, March 1998.

[111] Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 6,219,690, US Patent and Trademark Office, Washington, DC, April 2001.

[112] Snaman, W. E., and Thiel, D. W. The VAX/VMS distributed lock manager. *Digital Technical Journal 5* (September 1987), 29–44.

[113] Soules, C. A. N., Appavoo, J., Hui, K., Da Silva, D., Ganger, G. R., Krieger, O., Stumm, M., Wisniewski, R. W., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003), USENIX Association, pp. 141–154.

[114] Spraul, M. Re: RFC: patch to allow lock-free traversal of lists with insertion. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=100264675012867&w=2` [Viewed June 23, 2004], October 2001.

[115] Spraul, M. [rfc] 0/5 rcu lock update. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=108546407726602&w=2` [Viewed June 23, 2004], May 2004.

[116] Steiner, J. Re: [lse-tech] [rfc, patch] 1/5 rcu lock update: Add per-cpu batch counter. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=108551764515332&w=2` [Viewed June 23, 2004], May 2004.

[117] Stone, H. S., and Cocke, J. Computer architecture in the 1990s. *IEEE Computer* (September 1991), 30–38.

[118] Stone, J. S., Stone, H. S., Heidelberger, P., and Turek, J. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology Systems and Applications 1*, 4 (November 1993), 58–71.

[119] Swaminathan, S., Stultz, J., Vogel, J., and McKenney, P. E. Fairlocks – a high performance fair locking scheme. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems* (Cambridge, MA, USA, November 2002), pp. 246–251.

[120] Tay, Y. C. *Locking Performance in Centralized Databases.* Academic Press, 1987.

[121] Torvalds, L. Linux 2.5.43. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2` [Viewed June 23, 2004], October 2002.

[122] Torvalds, L. Linux 2.5.44. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=103500176112851&w=2 [Viewed June 23, 2004], October 2002.

[123] Torvalds, L. Linux 2.5.45. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=103602621711679&w=2 [Viewed June 23, 2004], October 2002.

[124] Torvalds, L. Linux 2.5.46. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=103645181102114&w=2 [Viewed June 23, 2004], November 2002.

[125] Torvalds, L. Linux 2.5.53. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=104070902324592&w=2 [Viewed June 23, 2004], December 2002.

[126] Torvalds, L. Linux 2.5.58. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=104252528009597&w=2 [Viewed June 23, 2004], January 2003.

[127] Torvalds, L. Linux 2.5.62. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=104552457430265&w=2 [Viewed June 23, 2004], February 2003.

[128] Torvalds, L. Linux 2.5.64. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=104683690231395&w=2 [Viewed June 23, 2004], March 2003.

[129] Torvalds, L. Linux 2.5.69. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105209603501299&w=2 [Viewed June 23, 2004], May 2003.

[130] Torvalds, L. Linux 2.5.70. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105400162802746&w=2 [Viewed June 23, 2004], May 2003.

[131] Torvalds, L. Linux 2.5.71. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105562576502426&w=2 [Viewed June 23, 2004], June 2003.

[132] Torvalds, L. Linux 2.5.73. Available: http://marc.theaimsgroup.com/?l=linux-kernel&m=105630824516148&w=2 [Viewed June 23, 2004], June 2003.

[133] Treiber, R. K. Systems programming: Coping with parallelism. RJ 5118, April 1986.

[134] Unrau, R., Krieger, O., Gamsa, B., and Stumm, M. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing 9*, 1/2 (1995), 345–370.

[135] Unrau, R., Stumm, M., Krieger, O., and Gamsa, B. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Tech. Rep. CSRI-268, University of Toronto, Toronto, Ontario, Canada, March 1992.

[136] Unrau, R. C. *Scalable Memory Management Through Hierarchical Symmetric Multiprocessing.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1993.

[137] Unrau, R. C., Krieger, O., Gamsa, B., and Stumm, M. Experiences with locking in a NUMA multiprocessor operating system. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (November 1994), USENIX Association, pp. 139–152.

[138] Unrau, R. C., Stumm, M., and Krieger, O. On the scalability of demand-driven parallel systems. In *Proceedings of the First International EURO-PAR Conference* (August 1995), LNCS 966, pp. 69–81.

[139] Vahalia, U. *UNIX Internals: The New Frontiers.* Prentice Hall, 1996.

[140] Weihl, W. E., and Liskov, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems 7*, 2 (July 1985), 244–269.

[141] Zhang, X., Casteneda, R., and Chan, E. W. Spin-lock synchronization on the Butterfly and KSR1. *IEEE Parallel and Distributed Technology Systems and Applications 2*, 1 (Spring 1994), 51–63.

Appendix A

Historical SMMP CPU Performance

Table A.1 presents the performance characteristics of Sequent CPUs over time. This data is also presented in Figure 1.1 on Page 2 and Figure 2.1 on Page 20.

Table A.1: Historical SMMP CPU Performance

					Memory Latency (us)	
Year	System Name	CPU	MHz	MIPS	Local	Remote
1984	B8000	NS32016	6	0.3	2.400	
1985	B8000	NS32016	10	0.5	1.500	
1986	B8000	NS32032	10	0.9	1.300	
1988	S2000	80386	16	4.0	1.000	
1989	S2000	80386	20	5.0	1.000	
1990	S2000	80486	25	20.0	1.000	
1991	S2000	80486	50	40.0	1.000	
1993	S2000	Pentium	60	60.0	1.000	
1994	S5000	Pentium	100	100.0	1.800	
1995	S5000	Pentium	166	166.0	1.800	
1996	NUMA-Q	P-Pro	180	180.0	0.450	5.00
1998	NUMA-Q	Xeon	360	360.0	0.250	2.50
1999	NUMA-Q	Tanner	450	450.0	0.250	2.50
2000	NUMA-Q	Cascades	700	700.0	0.225	2.50
2001	NUMA-Q	Cascades	900	900.0	0.225	2.50

Appendix B

Memory Ordering Issues

The discussion in Section 2.2.17 on Page 57 focussed on computer systems with sequential consistency. Atomic insertion poses additional problems on systems with very weak memory ordering, as noted in discussions on LKML [70]. This appendix focuses on these problems and some solutions.

Some of these problems may be addressed by using the `smp_wmb()` primitive as shown on line 9 of Figure B.1. This `smp_wmb()` guarantees that the element initialization in lines 6-8 is not executed before the element is added to the list on line 10. On many CPUs, this is sufficient, and the lock-free search on lines 14-26 will then operate correctly.

However, some CPUs, such as Alpha, have extremely weak memory ordering such that the code on line 20 of Figure B.1 could see the old garbage values that were present before the initialization on lines 6-8.

Figure B.2 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating caches lines are processed by the different partitions of the caches. Assume that the list header `head` will be processed by cache bank 0 and that the new element will be processed by cache bank 1. On Alpha, the `smp_wmb()` will guarantee that the cache invalidates performed by lines 6-8 of Figure B.1 will reach the interconnect before that of line 10 does, but makes absolutely no guarantee about the order in which the new values will reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is very busy, but cache bank 0 is idle. This could result in the cache invalidates for the new element being delayed, so that the reading CPU gets the new value for the pointer, but sees the old cached values for the new element. See HP/Compaq/DEC's Alpha documentation [19] for more information, or if you think

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     smp_wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         /* BUG ON ALPHA!!! */
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure B.1: Insert and Lock-Free Search

that I am just making all this up.

In the Linux kernel, this can be fixed in an implementation-independent manner by inserting an `smp_read_barrier_depends()` between the pointer fetch and dereference, as shown on line 19 of Figure B.3. However, this imposes unneeded overhead on systems (such as i386, IA64, PPC, and SPARC) that respect data dependencies on the read side. A `read_barrier_depends()` primitive has been added to the Linux 2.6 kernel to eliminate overhead on these systems [122]. Furthermore, the Linux list-manipulations APIs have been augmented by the addition of RCU variants that incorporate whatever memory barriers are required on a given CPU architecture, as shown in Figure 4.2 on Page 101.

It is also possible to implement a software barrier that could be used in place of `smp_wmb()`, which would force all reading CPUs to see the writing CPU's writes in order [69]. However, this approach was deemed by the Linux community to impose excessive overhead on extremely weakly ordered CPUs such as Alpha.[1] This software barrier could

[1]CPUs that respect data dependencies would define such a barrier to simply be `smp_wmb()`.

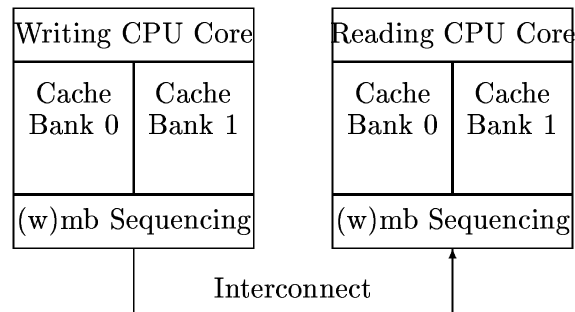


Figure B.2: Why smp_read_barrier_depends() is Required

be implemented by sending inter-processor interrupts (IPIs) to all other CPUs. Upon receipt of such an IPI, a CPU would execute a memory-barrier instruction, implementing a memory-barrier shootdown. Additional logic is required to avoid deadlocks.

Note that a memory-barrier shootdown could be used to eliminate the need for read-side memory barriers entirely, but at the cost of substantial overhead imposed on updates, though only on Alpha. A hardware implementation of memory-barrier shootdown might have substantial performance benefits in this case. Alternatively, given that HP has announced the Alpha CPU's end of life, perhaps it will soon be possible to ignore this entire issue, thereby simplifying RCU read-side code. Should this happen, the afore-mentioned RCU list-manipulation API could be adjusted to remove the read-side memory barriers.

Keir Fraser [27] suggests an alternative approach, where freed memory is initialized with a known pattern, and memory thus freed may not be reallocated until every CPU has executed an `smp_mb()` instruction. This approach works well when all memory is subject to lock-free search, but unnecessarily reduced cache warmth in other cases.

For more information on memory-consistency models, see Gharachorloo's exhaustive technical report [33].

```
 1 struct el *insert(long key, long data)
 2 {
 3     struct el *p;
 4     p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5     spin_lock(&mutex);
 6     p->next = head.next;
 7     p->key = key;
 8     p->data = data;
 9     smp_wmb();
10     head.next = p;
11     spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16     struct el *p;
17     p = head.next;
18     while (p != &head) {
19         smp_read_barrier_depends();
20         if (p->key == key) {
21             return (p);
22         }
23         p = p->next;
24     };
25     return (NULL);
26 }
```

Figure B.3: Safe Insert and Lock-Free Search

Appendix C

Additional RCU Implementations

Chapter 4 described a number of implementations of RCU infrastructure. This appendix describes a number of other implementations. Section C.1 elaborates on the induced-quiescent-state schemes described in Section 4.3 on Page 108, presenting *rcu-taskq*, which is a full implementation with batching and induced quiescent states that runs in the Linux kernel. Section C.2 elaborates on the leveraged-quiescent-state schemes described in Section 4.4 on Page 113, presenting the *rcu-krcud*, *X-rcu*, and *rcu-poll* RCU implementations. Finally, Section C.3 describes a Linux RCU infrastructure that permits preemption in the read-side critical sections, in a manner similar to K42's RCU infrastructure described in Section 4.4.3 on Page 130.

C.1 Induced Quiescent States With Batching (rcu-taskq)

Although simple inducing of quiescent states correctly implements RCU, the overhead of the context switches can be burdensome, especially when the update is a single simple linked-list modification. One way to reduce this burden on updates is batching, where a single set of context switches serves multiple updates, thereby amortizing the context-switch overhead. Although the `free_pending_rcus()` approach described in Section 4.3.2 on Page 111 showed how to implement batching, it did not describe when, how, and from where the `free_pending_rcus()` function should be invoked. This section presents one approach suitable for use within the Linux kernel.

Dipankar Sarma designed and implemented in the *rcu-taskq* patch, which uses a single task and a global set of callback queues. By coincidence, this implementation is similar to

one that was considered for DYNIX/ptx, but rejected as being insufficiently parallel. The task forces each of a set of per-CPU kernel daemons to schedule itself; when each done so, the grace period has expired. This implementation thus directly forces quiescent states, unlike the other implementations, which instead measure naturally occurring quiescent states. Its grace-period latency increases with increasing load on the system, as noted earlier, but is the only implementation with absolutely zero load on the system when there are no RCU callbacks in flight.

Figure C.1 shows the `call_rcu()` implementation. Lines 8-9 initialize the callback, lines 11 and 15 handle locking, lines 12-13 record the initial list state, and line 14 adds the callback to the `rcu_wait_list`. Lines 17-18 start the task if lines 12-13 found the list initially empty.

```
 1 void call_rcu(struct rcu_head * head,
 2               void (*func)(void * arg),
 3               void * arg)
 4 {
 5     unsigned long flags;
 6     int start = 0;
 7
 8     head->func = func;
 9     head->arg = arg;
10
11     spin_lock_irqsave(&rcu_lock, flags);
12     if (list_empty(&rcu_wait_list))
13         start = 1;
14     list_add(&head->list, &rcu_wait_list);
15     spin_unlock_irqrestore(&rcu_lock, flags);
16
17     if (start)
18         schedule_task(&rcu_task);
19 }
```

Figure C.1: *rcu-taskq* call_rcu() Implementation

The task started by `call_rcu()` invokes the function `process_pending_rcus()`, shown in Figure C.2. Lines 8-10 snapshot `rcu_wait_list` into a local list. Line 13 then invokes `wait_for_rcu()` to wait for a full grace period to elapse. Finally, lines 15-23 invoke the callbacks from the local list.

Figure C.3 shows `wait_for_rcu()`. Lines 6-10 awaken the *krcud* daemons for the other CPUs, and lines 11-13 wait for these daemons to respond.

Figure C.4 shows the code for the *krcud* daemons. Lines 6-20 initialize the daemon,

```
 1 static void process_pending_rcus(
 2                    void *arg)
 3 {
 4     LIST_HEAD(rcu_current_list);
 5     struct list_head * entry;
 6
 7     spin_lock_irq(&rcu_lock);
 8     list_splice(&rcu_wait_list,
 9                 rcu_current_list.prev);
10     INIT_LIST_HEAD(&rcu_wait_list);
11     spin_unlock_irq(&rcu_lock);
12
13     wait_for_rcu();
14
15     while ((entry = rcu_current_list.prev)
16               != &rcu_current_list) {
17         struct rcu_head * head;
18
19         list_del(entry);
20         head = list_entry(entry,
21                   struct rcu_head, list);
22         head->func(head->arg);
23     }
24 }
```

Figure C.2: *rcu-taskq* process_pending_rcus() Implementation

```
 1 static void wait_for_rcu(void)
 2 {
 3     int cpu;
 4     int count;
 5
 6     for (cpu = 0; cpu < smp_num_cpus; cpu++) {
 7         if (cpu == smp_processor_id())
 8             continue;
 9         up(&krcud_sema(cpu));
10     }
11     count = 0;
12     while (count++ < smp_num_cpus - 1)
13         down(&rcu_sema);
14 }
```

Figure C.3: *rcu-taskq* wait_for_rcu() Implementation

set its priority high, blocking signals, binding to the corresponding CPU, setting the task name, initializing the task name, and informing the `spawn_krcud()` task that the daemon is ready to process requests. Lines 22-26 process each request, alternately sleeping on the `krcud_sema` and waking up the `process_pending_rcus()` task.

```
 1 static int krcud(void * __bind_cpu)
 2 {
 3     int bind_cpu = *(int *) __bind_cpu;
 4     int cpu = cpu_logical_map(bind_cpu);
 5
 6     daemonize();
 7     current->policy = SCHED_FIFO;
 8     current->rt_priority = 1001 +
 9       sys_sched_get_priority_max(SCHED_FIFO);
10
11     sigfillset(&current->blocked);
12
13     /* Migrate to the right CPU */
14     set_cpus_allowed(current, 1UL << cpu);
15
16     sprintf(current->comm,
17             "krcud_CPU%d", bind_cpu);
18     sema_init(&krcud_sema(cpu), 0);
19
20     krcud_task(cpu) = current;
21
22     for (;;) {
23         while (down_interruptible(
24                 &krcud_sema(cpu)));
25         up(&rcu_sema);
26     }
27 }
```

Figure C.4: *rcu-taskq* krcud() Implementation

C.2 Further Leveraging of Quiescent States

The following sections describe the *rcu-krcud*, *X-rcu*, and *rcu-poll* RCU implementations. These are similar to the implementation in the Linux 2.6 kernel in that they leverage naturally occurring quiescent states, use per-CPU counters to record passage through these quiescent states, and provide a barrier computation that senses the quiescent-state counters to determine when the system has passed through a grace period. They differ primarily in how the barrier computation is implemented, *rcu-krcud* uses kernel daemons, *X-rcu* uses per-CPU timers, and *rcu-poll* uses tasklets but also interrupts CPUs to expedite

grace-period detection.

C.2.1 rcu-krcud

Dipankar Sarma also designed and coded the *rcu* patch,[1] which is also a counters-and-barrier implementation similar to that described in Section 4.4.1. This implementation is quite similar to the implementation in the Linux 2.6 kernel, differing only in that it uses per-CPU kernel daemons rather than the per-CPU clock interrupt handler to drive the RCU barrier computation. It therefore uses per-CPU queues of callbacks and context-switch counters instrumenting the quiescent states. However, it uses per-CPU kernel daemons to periodically check for the end of grace periods, which means that it cannot easily check for the CPU having been idle since running the kernel daemon by definition displaces idle-loop execution. In contrast, interrupt-driven barrier schemes can easily check to see if they have interrupted the idle loop. The per-CPU kernel daemons therefore require special care to ensure that idle-loop execution eventually results in a quiescent state, for example, by ensuring that the scheduler is invoked periodically whenever RCU callbacks are pending. These daemons are awakened by a timer that is scheduled only when there is at least one callback in the system. Dipankar Sarma implemented this variant to evaluate use of kernel daemons rather than architecture-dependent timer hooks.

The `call_rcu()` function simply constructs the callback, enqueues it onto the current CPU's `RCU_nxtlist`, then schedules the current CPU's tasklet, as shown in Figure C.5.

The scheduler is instrumented as shown in Figure C.6. As with *X-rcu*, this is a local increment without locking, atomic instructions, or cache thrashing, but, due to the lack of a per-CPU data area, array-indexing instructions are required.

The code that performs periodic RCU processing is shown in Figure C.7. Uniprocessor kernels invoke it directly from the timeout handler, while SMP kernels invoke it from the *krcud* daemons that are awakened by the timeout handler.

[1] To avoid confusion with the other RCU implementations, this dissertation calls this patch *rcu-krcud*.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5     int cpu = cpu_number_map(
 6                 smp_processor_id());
 7     unsigned long flags;
 8
 9     head->func = func;
10     head->arg = arg;
11     local_irq_save(flags);
12     list_add_tail(&head->list,
13                   &RCU_nxtlist(cpu));
14     local_irq_restore(flags);
15     tasklet_schedule(&RCU_tasklet(cpu));
16 }
```

Figure C.5: *rcu-krcud* call_rcu() Implementation

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3   prefetch(next);
4   prev->work.need_resched = 0;
5 +  RCU_qsctr(prev->cpu)++;
6
7   if (likely(prev != next)) {
8   rq->nr_switches++;
```

Figure C.6: *rcu-krcud* Scheduler Instrumentation

```
1 static void rcu_percpu_tick_common(void)
2 {
3     rcu_process_callbacks(0);
4 }
```

Figure C.7: *rcu-krcud* Timer Processing

C.2.2 X-rcu

Dipankar Sarma designed and coded the *X-rcu* implementation, which is similar to the implementation in the Linux 2.6 kernel described in Section 4.4.1, except that it uses per-CPU timers rather than the per-CPU `scheduler_tick()` clock interrupt to drive the RCU barrier computation. It thus uses a per-CPU context switch counter to instrument this quiescent state, uses per-CPU queues to track callbacks, and per-CPU timers to track quiescent states as needed to find the end of grace periods. The timers further check for running from idle, which is a second quiescent state. The purpose of this variant was to evaluate the use of timers rather than the kernel daemons or timer hooks used by the *rcu-krcud* and *rcu-ltimer* implementations. In the end, the advent of an architecture-independent timer hook in the 2.5 kernel made timer-interrupt hooks acceptable. These timer-interrupt hooks have the advantage that they are guaranteed to occur on each CPU on a regular basis, while the timers could potentially migrate to some other CPU.

The `call_rcu()` function constructs the callback and enqueues it onto the current CPU's `rcu_nextlist`, as shown in Figure C.8.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5         unsigned long flags;
 6
 7         head->func = func;
 8         head->arg = arg;
 9         local_irq_save(flags);
10         list_add_tail(&head->list,
11                 &this_cpu(rcu_nextlist));
12         local_irq_restore(flags);
13 }
```

Figure C.8: *X-rcu* call_rcu() Implementation

Figure C.9 shows how the scheduler is instrumented. The added line 5 compiles to a local increment, with no locking, atomic operations, or cache thrashing.

Figure C.10 shows the processing done by the per-CPU timer handler, currently set up to execute every 5 jiffies on each CPU. This code detects idle-loop execution and counts this as a quiescent state. It then invokes `rcu_process_callbacks()` to advance callbacks

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3          prefetch(next);
4          prev->work.need_resched = 0;
5 +        per_cpu(rcu_qsctr, prev->cpu)++;
6
7          if (likely(prev != next)) {
8                  rq->nr_switches++;
```

Figure C.9: *X-rcu* Scheduler Instrumentation

```
1 static void rcu_percpu_tick(void)
2 {
3         /* Check for idle loop */
4         if (task_idle(current))
5                 this_cpu(rcu_qsctr)++;
6         rcu_process_callbacks();
7 }
```

Figure C.10: *X-rcu* Timer Processing

as ends of grace periods are detected.

The *X-rcu* callback processing proceeds as shown in Figure C.11.

The `rcu_process_callbacks()` function shown in Figure C.12 handles the overall flow. Lines 3-12 move callbacks from `rcu_currlist` to `rcu_donelist` after the end of a grace period. Line 14 invokes `rcu_move_next_batch()` (shown in Figure C.13), which moves callbacks from `rcu_nextlist` to `rcu_currlist`, initiating grace-period detection if needed. Line 16 calls `rcu_check_quiescent_state()`, which checks to see if the current CPU has passed through a quiescent state since the beginning of the current grace period. Lines 18-22 call `rcu_invoke_callbacks()` to invoke any callbacks in `rcu_donelist`.

The `rcu_move_next_batch()` function shown in Figure C.13 disables local interrupts (line 3), and then checks to see if `rcu_currlist` is empty and `rcu_nextlist` is not (lines 4-7). If so, it moves the contents of `rcu_nextlist` to `rcu_currlist` (lines 8 and 9), then re-enables interrupts (line 12). It then obtains a new RCU batch number (lines 18-19) and registers it using `rcu_reg_batch()` (line 20, see Figure C.16 for this function's definition) under the `rcu_lock`.

If lines 4-5 find `rcu_currlist` to be nonempty, `rcu_move_next_batch()` simply re-enables interrupts and returns (line 23).

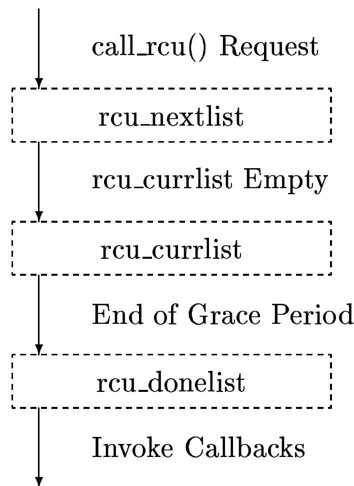


Figure C.11: RCU Callback Flow

```
 1 static void rcu_process_callbacks(void)
 2 {
 3     if (!list_empty(
 4           &this_cpu(rcu_currlist)) &&
 5         RCU_BATCH_GT(rcu_currbatch,
 6                    this_cpu(rcu_batch))) {
 7         list_splice(
 8             &this_cpu(rcu_currlist),
 9             &this_cpu(rcu_donelist));
10         INIT_LIST_HEAD(
11             &this_cpu(rcu_currlist));
12     }
13
14     rcu_move_next_batch();
15
16     rcu_check_quiescent_state();
17
18     if (!list_empty(
19             &this_cpu(rcu_donelist))) {
20         rcu_invoke_callbacks(
21             &this_cpu(rcu_donelist));
22     }
23 }
```

Figure C.12: *X-rcu* rcu_process_callbacks()

```
 1 static void rcu_move_next_batch(void)
 2 {
 3     local_irq_disable();
 4     if (!list_empty(
 5              &this_cpu(rcu_nextlist)) &&
 6         list_empty(
 7              &this_cpu(rcu_currlist))) {
 8         list_splice(&this_cpu(rcu_nextlist),
 9                     &this_cpu(rcu_currlist));
10         INIT_LIST_HEAD(
11                     &this_cpu(rcu_nextlist));
12         local_irq_enable();
13
14         /*
15          * start the next batch of callbacks
16          */
17         spin_lock(&rcu_lock);
18         this_cpu(rcu_batch) =
19                      rcu_currbatch + 1;
20         rcu_reg_batch(this_cpu(rcu_batch));
21         spin_unlock(&rcu_lock);
22     } else {
23         local_irq_enable();
24     }
25 }
```

Figure C.13: *X-rcu* rcu_move_next_batch()

The `rcu_check_quiescent_state()` function shown in Figure C.14 checks to see if the current CPU has gone through a quiescent state, and, if so, advances the barrier computation.

Lines 6-8 check to see if this CPU has already passed through a quiescent state during the current grace period, and, if so, line 6 simply returns. Lines 17-22 check to see if this is the first that this CPU has heard of the current grace period, and, if so, lines 19-20 take a snapshot of this CPU's context-switch counter in `rcu_last_qsctr` and returns. Lines 23-26 check to see if this CPU has passed through a quiescent state since the snapshot, and, if not, line 25 simply returns.

Execution reaches line 29 when this CPU first determines that it has passed through a quiescent state in the current grace period. Lines 28-44 publish this fact under the global `rcu_lock`, which possibly marks the end of the current grace period. Line 33 clears this CPU's bit in `rcu_cpumask`, which publicizes the fact that this CPU has passed through a quiescent state during the current grace period. Lines 34-35 set `rcu_last_qsctr` to an invalid quantity, which will indicate that this CPU is not yet aware of the next grace

period. If there are other CPUs that have not yet passed through their quiescent states, then lines 36-41 release the `rcu_lock` and return. Execution reaches line 42 if this CPU is the last one to detect that it has passed through a quiescent state during the current grace period, which marks the end of the grace period. Line 42 increments `rcu_currbatch`, which signals the end of the grace period. Line 43 invokes `rcu_reg_batch()` to initiate a new grace period if needed, and line 36 releases the `rcu_lock`.

Figure C.15 shows `rcu_invoke_callbacks()`, which simply loops through the list of callbacks, invoking each in turn.

Figure C.16 shows `rcu_reg_batch()`, which publicizes the beginning of a new grace period, if needed. Lines 4-7 check to see if the batch number of the requested grace period is larger than that of the largest-numbered grace period that has been requested thus far (the `RCU_BATCH_LT()` macro handles wraparound). If so, line 6 publicizes the new maximum batch number. If the largest-numbered grace period requested thus far has already completed or if a grace period is currently in progress, lines 8-12 simply return. Otherwise, line 13 sets `rcu_cpumask` to indicate that all CPUs need to pass through a quiescent state, which publicizes the start of a new grace period.

C.2.3 rcu-poll

The *rcu-poll* algorithm was designed and coded by Andrea Arcangeli and Dipankar Sarma. It appears in the "-aa" series of kernels and was the first Linux RCU variant to be used in production, appearing in recent SuSE releases of the Linux 2.4 kernel, starting with SuSE 7.3 Update, and is similar to the implementation used in VM/XA in the late 80s [39, 105]. Unlike the *X-rcu* and *rcu-krcud* algorithms, which use per-CPU lists of RCU callbacks, *rcu-poll* uses a single set of RCU-callback lists, which are processed by a single tasklet. This single list and tasklet results in *rcu-poll* being quite a bit simpler than most of the other implementations, but also results in higher grace-period-detection overhead, due to this single list being thrashed among the CPUs.

In addition, the CPUs are interrupted in order to induce them to enter the scheduler, so that this implementation combines inducing and observing quiescent states. This results in less batching than do the other implementations, but also results in extremely short

```
 1 static void rcu_check_quiescent_state(void)
 2 {
 3     int cpu = cpu_number_map(
 4                 smp_processor_id());
 5
 6     if (!test_bit(cpu, &rcu_cpumask)) {
 7         return;
 8     }
 9
10     /*
11      * May race with rcu per-cpu tick -
12      * in the worst case
13      * we may miss one quiescent state
14      * of that CPU. That is tolerable.
15      * So no need to disable interrupts.
16      */
17     if (this_cpu(rcu_last_qsctr) ==
18                     RCU_QSCTR_INVALID) {
19         this_cpu(rcu_last_qsctr) =
20                     this_cpu(rcu_qsctr);
21         return;
22     }
23     if (this_cpu(rcu_qsctr) ==
24             this_cpu(rcu_last_qsctr)) {
25         return;
26     }
27
28     spin_lock(&rcu_lock);
29     if (!test_bit(cpu, &rcu_cpumask)) {
30         spin_unlock(&rcu_lock);
31         return;
32     }
33     clear_bit(cpu, &rcu_cpumask);
34     this_cpu(rcu_last_qsctr) =
35                     RCU_QSCTR_INVALID;
36     if (rcu_cpumask != 0) {
37         /* All CPUs haven't gone
38            through a quiescent state */
39         spin_unlock(&rcu_lock);
40         return;
41     }
42     rcu_currbatch++;
43     rcu_reg_batch(rcu_maxbatch);
44     spin_unlock(&rcu_lock);
45 }
```

Figure C.14: *X-rcu* rcu_check_quiescent_state()

```
 1 static inline void rcu_invoke_callbacks(
 2                  struct list_head *list)
 3 {
 4     struct list_head *entry;
 5     struct rcu_head *head;
 6
 7     while (!list_empty(list)) {
 8         entry = list->next;
 9         list_del(entry);
10         head = list_entry(entry,
11                  struct rcu_head, list);
12         head->func(head->arg);
13     }
14 }
```

Figure C.15: *X-rcu* rcu_invoke_callbacks()

```
 1 static inline void rcu_reg_batch(
 2                rcu_batch_t newbatch)
 3 {
 4     if (RCU_BATCH_LT(rcu_maxbatch,
 5                         newbatch)) {
 6         rcu_maxbatch = newbatch;
 7     }
 8     if (RCU_BATCH_LT(rcu_maxbatch,
 9             rcu_currbatch) ||
10         (rcu_cpumask != 0)) {
11         return;
12     }
13     rcu_cpumask = cpu_online_map;
14 }
```

Figure C.16: *X-rcu* rcu_reg_batch()

average grace-period latencies.

The `call_rcu()` function constructs a callback, enqueues it onto a global `rcu_nxtlist`, then schedules the tasklet, as shown in Figure C.17.

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *arg),
 3               void *arg)
 4 {
 5     head->func = func;
 6     head->arg = arg;
 7
 8     spin_lock_bh(&rcu_lock);
 9     list_add(&head->list, &rcu_nxtlist);
10     spin_unlock_bh(&rcu_lock);
11
12     tasklet_hi_schedule(&rcu_tasklet);
13 }
```

Figure C.17: *rcu-poll* call_rcu() Implementation

The scheduler is instrumented in much the same way as for the previous algorithms, as shown in Figure C.18.

```
1 @@ -685,6 +686,7 @@
2  switch_tasks:
3          prefetch(next);
4          prev->work.need_resched = 0;
5 +        RCU_quiescent(prev->cpu)++;
6
7          if (likely(prev != next)) {
8                  rq->nr_switches++;
```

Figure C.18: *rcu-poll* Scheduler Instrumentation

Periodic RCU processing is handled by a single tasklet, whose body is shown in Figure C.19. This tasklet invokes `rcu_prepare_polling()` to snapshot each CPU's quiescent state counters if polling is not yet in progress and if there are pending callbacks. If polling has already been started, it instead invokes `rcu_polling()` to check to see if the grace period has ended. This ensures all CPUs have passed through their quiescent states via the context switch.

```
 1 static void rcu_process_callbacks(
 2                   unsigned long data)
 3 {
 4     int stop;
 5
 6     spin_lock(&rcu_lock);
 7     if (!rcu_polling_in_progress)
 8         stop = rcu_prepare_polling();
 9     else
10         stop = rcu_polling();
11     spin_unlock(&rcu_lock);
12
13     if (!stop)
14         tasklet_hi_schedule(&rcu_tasklet);
15 }
```

Figure C.19: *rcu-poll* Tasklet Body

The *rcu-poll* callback processing is initiated by the `rcu_prepare_polling()` function, shown in Figure C.20. This function relies on `rcu_process_callbacks()` (see Figure C.19) acquiring the `rcu_lock`. Lines 12-27 check to see if there are callbacks waiting in `rcu_nxtlist`, and, if so, starts a grace period. Lines 13-14 move the list from `rcu_nxtlist` to `rcu_curlist`. Line 16 records the fact that a grace period is now in progress. Lines 18-25 mark each CPU (other than the current one) as needing to go through a quiescent state, take a snapshot of each CPU's context-switch counter, and expedite a context switch. Line 26 indicates that grace-period polling needs to continue–if `rcu_nxtlist` had been empty, polling would cease until the next `call_rcu()` invocation.

Figure C.21 shows the `rcu_polling()` function. Lines 6-13 check each CPU that has not yet been observed passing through a quiescent state (as indicated by the `rcu_qsmask` check at line 9) to see if that CPU's `RCU_quiescent` counter has advanced since the `rcu_prepare_polling()` started the current grace period. If it has, then that CPU has recently passed through a quiescent state, so line 12 clears its bit from `rcu_qsmask`. Line 16 then checks to see if all CPUs have now passed through their quiescent states. If so, line 17 invokes `rcu_completion()` to mark the end of the grace period. If another grace period is required, `rcu_completion` will have started it, and will then return zero to signal that grace-period polling should continue.

Figure C.22 shows the `rcu_completion()` function that is invoked at the end of a grace period. Line 5 records the fact that a grace period is no longer in progress, line 6 invokes

```
 1 static int rcu_prepare_polling(void)
 2 {
 3     int stop;
 4     int i;
 5
 6 #ifdef DEBUG
 7     if (!list_empty(&rcu_curlist))
 8         BUG();
 9 #endif
10
11     stop = 1;
12     if (!list_empty(&rcu_nxtlist)) {
13         list_splice(&rcu_nxtlist, &rcu_curlist);
14         INIT_LIST_HEAD(&rcu_nxtlist);
15
16         rcu_polling_in_progress = 1;
17
18         for (i = 0; i < smp_num_cpus; i++) {
19             int cpu = cpu_logical_map(i);
20
21             rcu_qsmask |= 1UL << cpu;
22             rcu_quiescent_checkpoint[cpu] =
23                          RCU_quiescent(cpu);
24             force_cpu_reschedule(cpu);
25         }
26         stop = 0;
27     }
28
29     return stop;
30 }
```

Figure C.20: *rcu-poll* rcu_prepare_polling()

```
 1 static int rcu_polling(void)
 2 {
 3     int i;
 4     int stop;
 5
 6     for (i = 0; i < smp_num_cpus; i++) {
 7         int cpu = cpu_logical_map(i);
 8
 9         if (rcu_qsmask & (1UL << cpu))
10             if (rcu_quiescent_checkpoint[cpu]
11                   != RCU_quiescent(cpu))
12                 rcu_qsmask &= ~(1UL << cpu);
13     }
14
15     stop = 0;
16     if (!rcu_qsmask)
17         stop = rcu_completion();
18
19     return stop;
20 }
```

Figure C.21: *rcu-poll* rcu_polling()

`rcu_invoke_callbacks()` to invoke the callbacks, and line 8 starts a new grace period, if required.

```
 1 static int rcu_completion(void)
 2 {
 3     int stop;
 4
 5     rcu_polling_in_progress = 0;
 6     rcu_invoke_callbacks();
 7
 8     stop = rcu_prepare_polling();
 9
10     return stop;
11 }
```

Figure C.22: *rcu-poll* rcu_completion()

Figure C.23 shows the `rcu_invoke_callbacks()` function. This is similar to that shown for *X-rcu* in Figure C.15, but processes a single global list rather than a per-CPU list, and removes elements from the list in a slightly different manner.

```
 1 static void rcu_invoke_callbacks(void)
 2 {
 3     struct list_head *entry;
 4     struct rcu_head *head;
 5
 6 #ifdef DEBUG
 7     if (list_empty(&rcu_curlist))
 8         BUG();
 9 #endif
10
11     entry = rcu_curlist.prev;
12     do {
13         head = list_entry(entry,
14                  struct rcu_head, list);
15         entry = entry->prev;
16
17         head->func(head->arg);
18     } while (entry != &rcu_curlist);
19
20     INIT_LIST_HEAD(&rcu_curlist);
21 }
```

Figure C.23: *rcu-poll* rcu_invoke_callbacks()

C.3 Preemption in Linux RCU (rcu-preempt)

Preemption was added to Linux in the 2.5.4 kernel. The addition of preemption means that read side kernel code is subject to involuntary context switches. If not taken into account, this leads to premature flagging of the ends of grace periods. There are two ways to handle preemption: (1) explicitly disabling preemption over read side code segments, and (2) considering only voluntary context switches to be quiescent states.

Explicitly disabling preemption over read side code segments adds unwanted overhead to reading processes, and removes some of the latency benefits provided by preemption. In contrast, considering only voluntary context switches to be quiescent states allows the kernel to reap the full benefit of reduced latency. Unfortunately, it also results in the possibility of unbounded grace-period durations, which eventually resulted in it not being incorporated into the Linux kernel. Nonetheless, in the spirit of full disclosure, this section examines the voluntary-context-switch option and its consequences.

The scheme for tracking only voluntary context switches is inspired by the K42 implementation [30], and was designed and implemented by Dipankar Sarma. K42's extensive use of blocking locks and short-lived threads results in use of thread termination rather than voluntary context switch as the K42 quiescent state (and also forestalls the possibility of unbounded grace-period lengths in K42). In addition, Linux migrates preempted tasks to other CPUs, which requires special tracking of these preempted tasks since their last voluntary context switch.

Dipankar Sarma created a prototype preemptible algorithm that is similar to *rcu-krcud*,[2] but adds per-CPU counts of preempted tasks, which operate in a manner in some ways similar to the generation mechanism in K42 [30]. The key concept is that a preemptible kernel must track tasks rather than CPUs. However, to avoid potentially expensive scans of the task list or the runqueues, the tasks are tracked on a per-CPU basis. When a task returns from a voluntary context switch (or is created), it is implicitly associated with the CPU that it starts running on. No matter how many times the task

[2]However, as noted earlier, this preemptible version of *rcu-krcud* has greatly reduced CPU overhead when there are no RCU callbacks in the system.

is preempted, from an RCU perspective, it remains affiliated with that CPU, even if it is migrated to other CPUs. Once it performs a voluntary context switch, it gives up its affiliation.

However, no additional work is done (over that done by a non-preemptible kernel running a non-preemptible implementation of RCU) until that task is preempted. The task then increments a per-CPU counter, which remains incremented until the task executes a voluntary context switch, possibly by exiting. The task then decrements that same per-CPU counter, even if the task is running on some other CPU at the time.

Of course, if there is a lot of preemption, it might be that a particular CPU *always* has at least one preempted task affiliated with it. However, the end of a grace period is marked not by the absence of tasks, but by each of the tasks that was either running or preempted at the start of the grace period having either exited or voluntarily switched context.

This distinction is maintained by providing each CPU with a pair of counters, a "next" counter that is incremented by tasks returning from their voluntary context switch onto the corresponding CPU, and a "current" counter that is only decremented. Note that the "next" counter will be also decremented whenever a task resumes execution quickly enough after being preempted. The end of the grace period occurs when all CPUs' "current" counters reach zero.[3] The roles of the counters in each pair are now reversed in order to start the next grace period, just after the base *rcu-krcud* portion of the algorithm moves the callbacks in the `rcu_nextlist` to `rcu_currlist`.

Each CPU's pair of counters is as shown in Figure C.24, along with the pair of pointers that handle the reversing of their roles. The `next_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that is atomically incremented (by a new `rcu_preempt_get()` function) when task affiliated with this CPU is preempted for the first time since its preceding voluntary context switch. The task records this pointer in a new `cpu_preempt_cntr`

[3] Unless one of the CPUs has been running a task continuously since before the start of the grace period, but this case is handled by the base *rcu-krcud* portion of the implementation.

pointer in its task structure, which is initially NULL. After the task resumes and voluntarily relinquishes the CPU[4], it atomically decrements the counter pointed to by its `cpu_preempt_cntr`, using a new `rcu_preempt_put()` function, and then NULLs out its `cpu_preempt_cntr` pointer.

```
1 extern atomic_t
2    rcu_preempt_cntr[2] __per_cpu_data;
3 extern atomic_t
4    *curr_preempt_cntr __per_cpu_data;
5 extern atomic_t
6    *next_preempt_cntr __per_cpu_data;
```

Figure C.24: rcu_preempt Per-CPU Counters

The `curr_preempt_cntr` pointer points to the element of `rcu_preempt_cntr[]` that `next_preempt_cntr` does not point to. This element of the array contains the number of tasks affiliated with this CPU that were first preempted before the beginning of the current grace period, and that must resume and voluntarily relinquish a CPU before the current grace period can expire. When this CPU becomes aware of the end of the current grace period, it exchanges the values of `next_preempt_cntr` and `curr_preempt_cntr`, so that the elements of the `rcu_preempt_cntr[]` array exchange roles.

The rest of the callback processing is very similar to that of the *rcu-krcud* algorithm. The major difference is that `rcu_check_quiescent_state()` must check that all tasks preempted on this CPU prior to the current grace period have voluntarily relinquished the CPU.

This implementation is currently not used in Linux for two reasons: (1) the grace periods can be arbitrarily long in a busy system with a low-priority task, and (2) the situations where it was thought useful turned out to have better solutions.

[4] Possibly after having been preempted several more times along the way. This is why the counter cannot be decremented immediately when the task is resumed, but must instead wait for the task to voluntarily relinquish the CPU.

Appendix D

RCU Performance on Large Hash Tables

Section 2.2.4 on Page 21 introduced a hash-table mini-benchmark that is used to compare the performance of a selected set of locking primitives. In Section 8.1 on Page 269, this set was expanded to include RCU.

However, all of the measurements were taken on a very small 32-bucket hash table, which is small enough to fit entirely into CPU cache. This appendix looks at the performance on larger hash tables that do not fit into CPU cache. Aside from the increased size (16,384 buckets instead of 32), the experiments are identical.

Performance on a read-only workload is shown in Figure D.1. Searches of this larger hash table incur greater capacity miss rates, increasing the overhead incurred in the critical section, thereby increasing scaling for the locked searches. However, this increased scaling has not come for free, as the overall performance has decreased by a factor of three. RCU still outperforms the next best mechanism by more than 30%.

Figure D.2 shows the performance of the same hash-table configuration with a mixed workload running on 4 CPUs, varying from read-only on the left-hand side to write-only on the right-hand side. This workload has short grace periods, so that about ten operations are completed per grace period. RCU remains optimal below about 15% writes.

Figure D.3 is similar to the preceding figure, but with longer grace periods so that there are about 100 operations per grace period. Under these conditions, RCU remains optimal with over 20% writes.

This data demonstrates that RCU retains significant performance advantages when used on large data structures that do not fit into CPU caches.

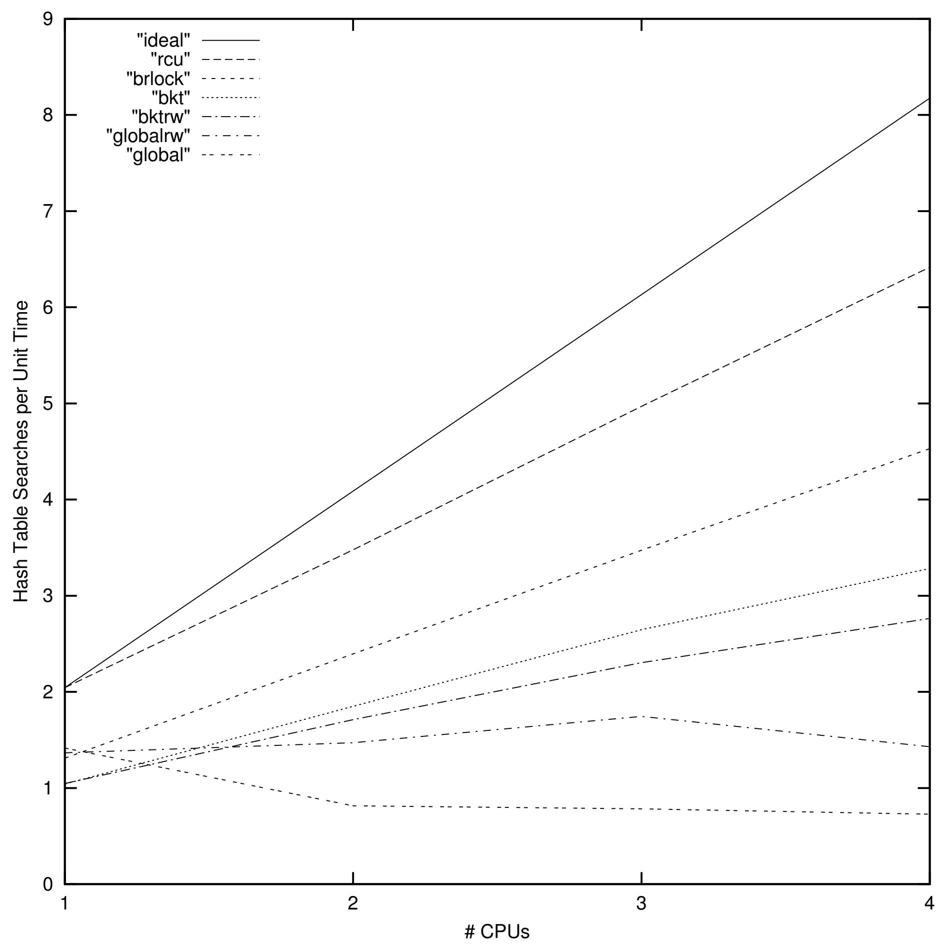


Figure D.1: Large Hash Table Performance for Read-Only Workload

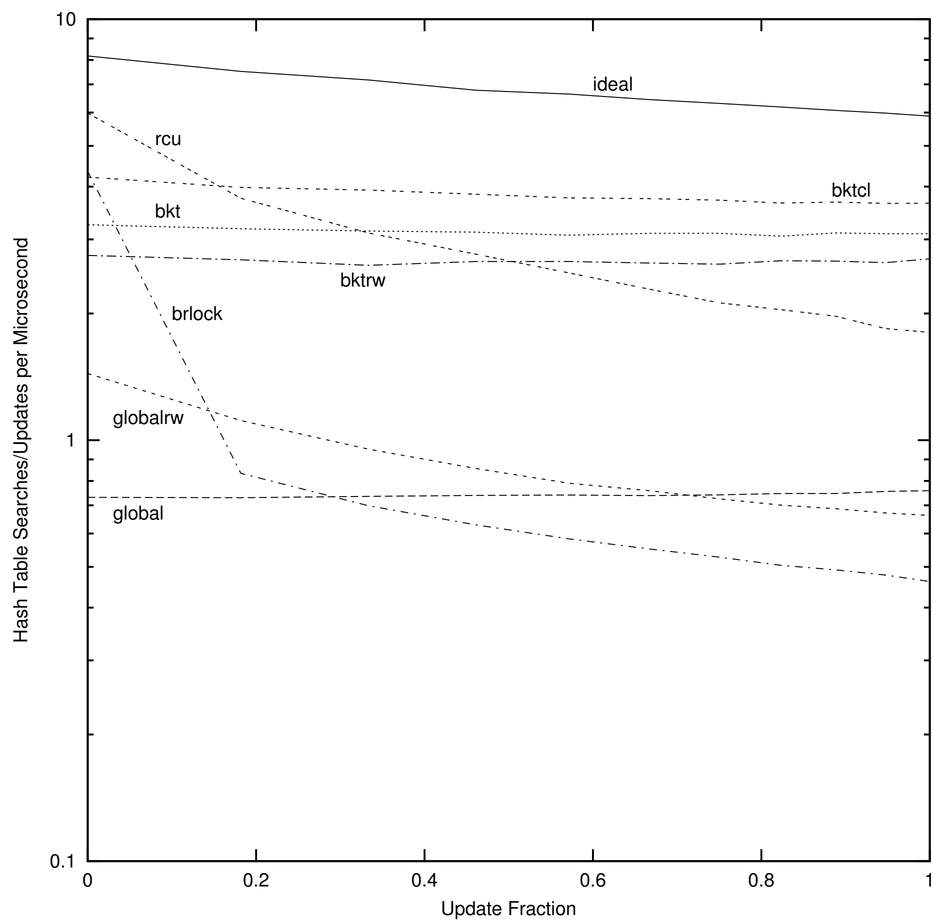


Figure D.2: Large Hash Table Performance for Mixed Workload and Short Grace Period

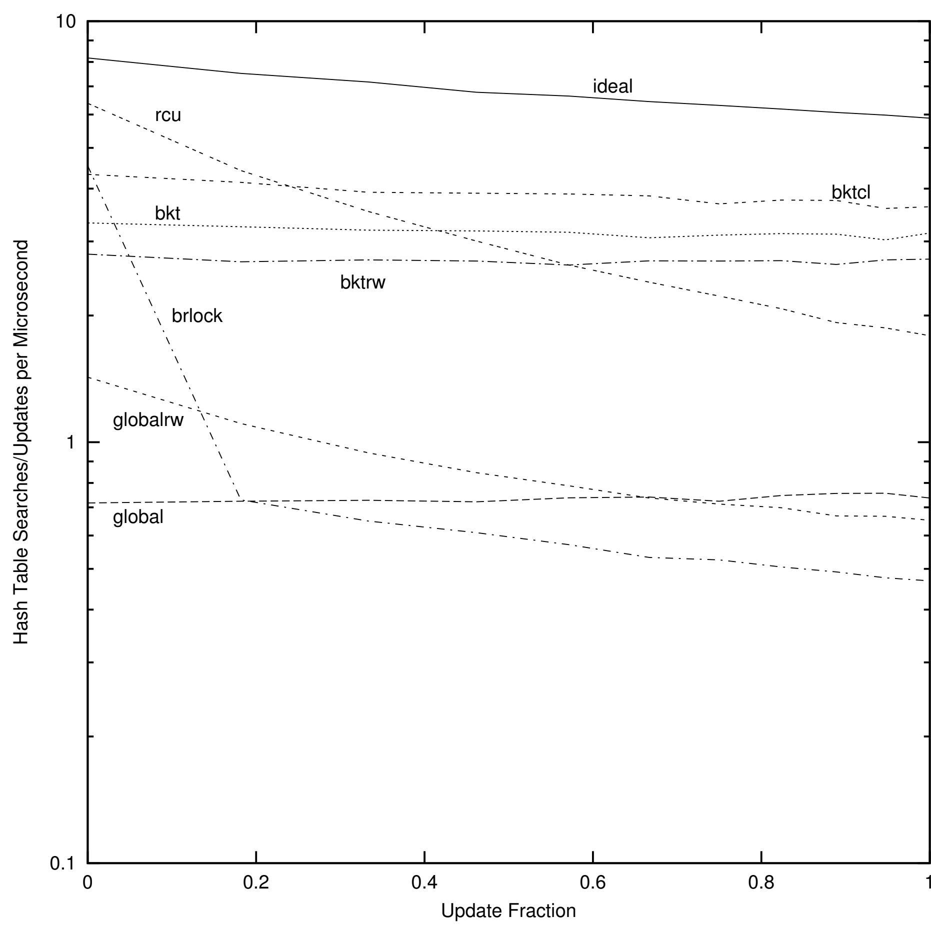


Figure D.3: Large Hash Table Performance for Mixed Workload and Long Grace Period

Biographical Note

Paul E. McKenney was born in Port Townsend, WA, USA on October 31, 1958. He received B.S. degrees in Mechanical Engineering and in Computer Science from Oregon State University in 1981, and an M.S. degree in Computer Science, also from Oregon State University with some coursework at Stanford University, in 1988.

Paul's interests include SMP and NUMA algorithms, as well as intellectual property issues, the latter less out of choice than through necessity.

From 1977 through 1980, Paul worked as a student programmer at the Oregon State University Computer Center, maintaining the student housing assignment and billing program, first in FORTRAN-II, later converting it to COBOL. Upon graduation in 1981 through 1985, Paul was a self-employed contract programmer in Corvallis, OR, working primarily on soft-realtime embedded systems. In 1986, Paul joined SRI International in the San Francisco Bay Area, where he worked as a Unix systems administrator on a Pyramid 90x, and later as a packet radio and Internet researcher.

In 1990, faced with the reality of a wife and two children in a two-bedroom apartment, Paul moved back to Oregon, joining Sequent Computer Systems, working first as a computer communications performance analyst, and later on SMP and NUMA algorithms in Sequent's DYNIX/ptx operating system kernel. In 1999, Sequent merged into IBM, and Paul became an IBM Distinguished Engineer. His focus shifted to AIX and later to Linux, after which he joined IBM's Storage Software Architecture team, though he still worked with the Linux kernel. Later, Paul returned to working full-time with Linux at IBM's Linux Technology Center.

Paul is a member of the Association for Computing Machinery, the Society of Automotive Engineers, and is a Senior Member of the Institute for Electrical and Electronics

Engineers. He received a second place award at the OGI CSE Student Research Symposium in 2001, and is a member of the IBM Academy of Technology.

Book Chapters.

1. McKenney, Paul E. *Pattern Languages of Program Design*, vol. 2. Addison-Wesley, June 1996, ch. 31 Selecting Locking Designs for Parallel Programs, pp. 501-531. The material in this chapter has been incorporated into a number of training programs from diverse companies, both in the area of pattern languages and in the area of parallel programming.

Journal Articles.

1. Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Da Silva, D. M., Krieger, O., Auslander, M. A., Edelsohn, D. J., Gamsa, B., Ganger, G. R., McKenney, Paul E., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal 42*, 1 (January 2003), 60–76. Describes how autonomic system software can be constructed using a hot-swapping approach based on read-copy update. `http://www.research.ibm.com/journal/sj/421/appavoo.pdf`

2. McKenney, Paul E., Slingwine, Jack, and Krueger, Phil. Experience With an Efficient Parallel Kernel Memory Allocator. *Software–Practice and Experience*, 31(2):235–257 (2001). Revision of 1993 paper, adding measurements of allocator performance in production use and a bit of NUMA analysis.

3. McKenney, Paul E. Differential Profiling. *Software–Practice and Experience*, 29(3):219-234 (1999) Revision of 1995 paper.

4. McKenney, Paul E. Selecting locking primitives for parallel programs. *Communications of the ACM 39*, 10 (October 1996), 75–82.

5. McKenney, Paul E. and Dove, Ken F. Efficient Demultiplexing of Incoming TCP Packets. *Computing Systems*, 5(2):141-157 (1992).

6. McKenney, Paul E. Stochastic Fairness Queuing. *Internetworking: Research and Experience*, 2:113-131 (1991).

7. McKenney, Paul E. and Bausbacher, Peter E. Physical- and Link-Layer Modeling of Packet-Radio Network Performance. *IEEE Journal on Selected Areas in Communications*, 9(1):59–64 (1991).

8. Lewis, T. G., Spitz, K. R., and McKenney, Paul E. An Interleave Principle for Demonstrating Concurrent Programs. *IEEE Software*, 1(10):54–64 (1984). October 1984, with Ted G. Lewis and Keith R. Spitz.

Refereed Conference/Workshop Articles.

1. McKenney, Paul E. and Sarma, D. Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications. To appear in *USENIX UseLinux* (June 2004).

2. McKenney, Paul E. RCU vs. Locking Performance on Different CPUs. In *linux.conf.au* (January 2004). Presents performance tradeoffs between locking and RCU on different CPU architectures.

3. Arcangeli, A., Cao, M., McKenney, Paul E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), pp. 338–367. Shows 12x performance improvement for System V IPC microbenchmark, and compares RCU infrastructures. I was lead author, the authors are listed in alphabetical order.

4. Swaminathan, S., Stultz, J., Vogel, J., and McKenney, Paul E. Fairlocks–a high performance fair locking scheme. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems* (Cambridge, MA, USA, November 2002), pp. 246–251. Presents a simple NUMA-aware locking scheme that avoids lock starvation.

5. McKenney, Paul E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. Read-copy update. In *Ottawa Linux Symposium* (June 2002),

pp. 338–367. Compares and contrasts a number of Linux read-copy-update implementations developed since the 2001 paper, one of which was accepted into the Linux 2.5.43 kernel in October 2002.

6. McKenney, Paul E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Described Linux implementations of the read-copy update techniques described in the PDCS'98 paper.

7. McKenney, Paul E. Practical performance estimation on shared-memory multiprocessors. In *Parallel and Distributed Computing and Systems* (Boston, MA, November 1999), pp. 125–134. Method of quickly estimating performance with reasonable accuracy. Variants of this can be used during design, before either code, compilers, or hardware is available.

8. McKenney, Paul E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518. Novel set of synchronization primitives that can result in near-zero synchronization overhead in restricted, but commonly occurring, situations. In more specialized cases, this set of primitives can result in negative overheads. Read-copy update was used in production in Sequent's DYNIX/ptx operating system in 1993, and was accepted into the Linux 2.5 kernel in October of 2002.

9. McKenney, Paul E. Differential profiling. In *MASCOTS'95* (Toronto, Canada, January 1995). Describes technique for locating scaling problems that are otherwise difficult to locate.

10. McKenney, Paul E., and Slingwine, J. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings* (Berkeley CA, February 1993). This paper demonstrated a memory allocator that could run faster on shared-memory parallel machines than previous allocators could on single-CPU

machines. This paper has been studied in a number of high-level college courses, and is cited by "UNIX Internals: The New Frontiers" by Uresh Vahalia.

11. McKenney, Paul E. and Dove, Ken F. Efficient Demultiplexing of Incoming TCP Packets. In *SIGCOMM'92 Conference Proceedings, Communications Architectures and Protocols*, (Baltimore, MD, USA, August 1992). This paper influenced the design of IPv6, and is cited by volume 2 of "TCP/IP Illustrated" by Wright and Stevens and "Gigabit Networking" by Craig Partridge.

12. McKenney, Paul E. and Graunke, Gary. Efficient Buffer Allocation on Shared-Memory Multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, (Tucson, AZ, USA, February 1992).

13. McKenney, Paul E. Stochastic Fairness Queuing. In *IEEE INFOCOM'90 Proceedings*, (San Francisco, CA, USA, June 1990) This paper has been cited by many computer-communications-network-traffic-control papers, and is still cited by new work in this area. It is also cited by the "Gigabit Networking" by Craig Partridge.

14. Shacham, Nachum and McKenney, Paul E. Packet Recovery in High-Speed Networks Using Coding and Buffer Management. In *IEEE INFOCOM'90 Proceedings*, (San Francisco, CA, USA, June 1990) This was the first paper that contained a realistic analysis of the costs and benefits of forward error correction (FEC) for recovering from congestion-induced packet losses. It is frequently referenced by ongoing work in this area, and is cited by "Gigabit Networking" by Craig Partridge.

15. McKenney, Paul E. and Bausbacher, Peter E. Physical- and Link-Layer Modeling of Packet-Radio Network Performance. In *MILCOM'90 Proceedings*, (Monterey, CA, USA, October 1990).

16. Beyer, David A., Frankel, Michael S., Hight, John M., Lee, Diane S., Lewis, Mark G., McKenney, Paul E., Naar, Jacques, Ogier, Richard G., Shacham, Nachum, and Zaumen, William T. Packet Radio Network Research, Development, and Application. In *MILCOM'89 Proceedings*, 1989.

17. McKenney, Paul E. High-Speed Event Counting and Classification Using a Dictionary Hash Technique. In Proceedings of the 1989 International Conference on Parallel Processing, (University Park, PA, USA, August 1989).

Unrefereed Journal Articles.

1. McKenney, Paul E., Sarma, Dipankar, and Soni, Maneesh. Scaling dcache with RCU. *Linux Journal* (January 2004).

2. McKenney, Paul E. Using RCU in the Linux 2.6 Kernel. *Linux Journal*, (October 2003). Shows empirical data comparing RCU with other locking mechanisms. Contains the infamous "old man vs. the brat" cartoon, compliments of my daughter, Melissa.

3. McKenney, Paul E. Benchmark Standards/Metrics and Capacity Planning. *UniForum*, 1994.

4. McKenney, Paul E. Congestion Avoidance. *;login: the USENIX Association Newsletter*, 13(4):9-12 (1988).

5. McKenney, Paul E. Broadcast Storms, Nervous Hosts, and Load Imbalances. *;login: the USENIX Association Newsletter*, 13(5):9-17 (1988).

6. Bailey, Kirk A., Boynton, Lee, McKenney, Paul E., Oliver, Gary J., Regan, David. User Defined Files. *Operating Systems Review*, (Fall 1981).

White Papers and Technical Reports.

1. Denny, B. and McKenney, Paul E. Experiment Design for CATE. SRI Technical Note ITAD-8600-TR-93-74, SRI International, March 1993.

2. Denny, B., Lee, D., and McKenney, Paul E. Traffic Generator Software Release Notes. SRI Technical Note ITAD-8600-TN-93-28, SRI International, February 1993.

3. McKenney, Paul E. Position Estimation. Oregon State University technical report, 1988. OSU Master's Project. Described method of calibrating acoustic-navigation transponder arrays. Prior to GPS, this was the preferred method of accurately determining the position of surface ships used both to conduct SONAR searches for salvage purposes and to map gravitational and geomagnetic data for mineral-exploration purposes. It is still the method of choice in many cases for submersibles below the depths penetrated by GPS signals.

Letters to the Editor.

1. McKenney, Paul E. Hope for Educational Sea Change. *Communications of the ACM*, 44(6):12, June 2001.

2. McKenney, Paul E. Mastering the Basics the Highest Priority. *Communications of the ACM*, 43(3):14, March 2000.

3. McKenney, Paul E. Renaissance Reflections. *Scientific American*, 280(5):10 May 1999.

Patents.

1. McKenney, P. E. Adaptive Reader-Writer Lock. Tech. Rep. US Patent 6,678,772, US Patent and Trademark Office, Washington, DC, January 2004. Dynamically switches between different locking primitives as needed to optimize performance.

2. McKenney, P. E. High speed counters. Tech. Rep. US Patent 6,668,310, US Patent and Trademark Office, Washington, DC, December 2003. Combine the "target" and "value" fields in a distributed reference counter in order to be able to manipulate them atomically with a single instruction.

3. Kingsbury, B. A., Sulmont, J. C., and McKenney, P. E. Message passing using shared memory of a computer. Tech. Rep. US Patent 6,629,152, US Patent and Trademark Office, Washington, DC, September 2003. Use of atomic operations to

implement a high-speed memory-based message-passing facility for shared-memory multiprocessor computers.

4. McKenney, P. E., and Krueger, P. E. Using hardware counters to estimate cache warmth for process/thread schedulers. Tech. Rep. US Patent 6,615,316, US Patent and Trademark Office, Washington, DC, September 2003. Markov models to estimate cache warmth given the performance-counter data available on commodity microprocessors.

5. Kingsbury, B. A., Casper, C., Krueger, P. E., and McKenney, P. E. User specifiable allocation of memory for processes in a multiprocessor computer having a non-uniform memory architecture. Tech. Rep. US Patent 6,505,286, US Patent and Trademark Office, Washington, DC, January 2003.

6. McKenney, P. E., Closson, K. A., and Malige, R. Lingering locks with fairness control for multi-node computer systems. Tech. Rep. US Patent 6,480,918, US Patent and Trademark Office, Washington, DC, November 2002.

7. Jackson, B. J., McKenney, P. E., Rajamony, R., and Rockhold, R. L. Scalable interruptible queue locks for shared-memory multiprocessor. Tech. Rep. US Patent 6,473,819, US Patent and Trademark Office, Washington, DC, October 2002.

8. McKenney, P. E., and Pulamarsetti, C. Optimized function execution for a multiprocessor computer system. Tech. Rep. US Patent 6,418,517, US Patent and Trademark Office, Washington, DC, July 2002.

9. Krueger, P. E., Casper, C., Dove, K. F., Kingsbury, B. A., and McKenney, P. E. Multiprocessor computer system with user specifiable process placement. Tech. Rep. US Patent 6,247,041, US Patent and Trademark Office, Washington, DC, June 2001.

10. Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor

system utilizing execution history and thread monitoring. Tech. Rep. US Patent 6,219,690, US Patent and Trademark Office, Washington, DC, April 2001. "Change in mode" aspect of RCU.

11. Kingsbury, B. A., Casper, C., Krueger, P. E., and McKenney, P. E. User specifiable allocation of memory for processes in a multiprocessor computer having a non-uniform memory architecture. Tech. Rep. US Patent 6,205,528, US Patent and Trademark Office, Washington, DC, March 2001.

12. Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,727,209, US Patent and Trademark Office, Washington, DC, March 1998. "Read-copy-update" aspect of RCU.

13. Slingwine, J. D., and McKenney, P. E. Method for maintaining data coherency using thread activity summaries in a multicomputer system. Tech. Rep. US Patent 5,608,893, US Patent and Trademark Office, Washington, DC, March 1997. "Synchronize data in different computers" aspect of RCU.

14. Slingwine, J. D., and McKenney, P. E. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Tech. Rep. US Patent 5,442,758, US Patent and Trademark Office, Washington, DC, August 1995. "Summary of execution history and thread monitoring infrastructure" aspect of RCU.

22 additional patent applications pending.