NTNU – Trondheim
Norwegian University of
Science and Technology

Analyzing Contextual Bias of Program Execution on Modern CPUs

Lars Kirkholt Melhus

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Rune E. Jensen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Variations in execution context has been shown to affect performance of programs on recent CPUs. Previous work has looked at offsetting the stack and changing link order – biasing performance measurements towards certain configurations. Variables such as the placement of stack, heap and text segments in memory, combined with memory access instructions, can impact program performance quite significantly.

The goal of this project is to model some of these effects for common use cases, and methods for avoiding them, in order to avoid bias and achieving peak performance. These effects might be highly platform dependent, so to limit the scope we will focus on a particular architecture, the Intel Core i7 “Ivy Bridge”. Case studies will include small isolated programs, and ideally also “real” applications such as FFT algorithms.

Abstract

Seemingly innocuous properties of the environment, such as the contents of system environment variables, or different link orders, can impact the performance of computer programs. Variations in external properties like these can *bias* programs towards certain configurations. These effects have been shown to be a significant issue in performance analysis, but unpredictable and difficult to deal with.

This thesis focuses on the underlying reasons for bias effects that can be experienced for example by changing environment variables, or using different link orders. Both of these factors can lead to differences in memory layout of either code or data, which in turn interacts with various hardware mechanisms. Through experimentation and careful measurements using performance counters, we identify several potential sources of bias on the Intel Core i7 “Ivy Bridge” architecture. Limitations imposed by the Loop Stream Detector is revealed, along with effects from 4K address aliasing. We show that bias is in fact not completely unpredictable, and discuss measures for avoiding it.

Our case studies show that even highly optimized Fourier transform and linear algebra libraries are prone to bias. We find that stack alignment significantly affects the performance of FFTW, and that in some cases more than 30 % performance gain can be made by avoiding address aliasing in ATLAS’ matrix-vector multiplication. Our results show that an awareness of program layout in memory is important, especially for users and developers of performance critical software.

Sammendrag

Ytelsesvurdering av programvare er påvirket av mange forskjellige, og tilsynelatende uviktige, egenskaper ved maskinen man gjør målinger på. For eksempel kan innholdet i miljøvariabler, eller rekkefølgen programfiler lenkes, være signifikant. Variasjoner i ytelse som følge av eksterne egenskaper som dette introduserer *bias*. Slike effekter kan være signifikante, men er også funnet å være vanskelige å forutse eller motvirke.

Denne oppgaven dreier seg om å studere underliggende årsaker til ytelsesforskjeller som kan forekomme for eksempel av å endre miljøvariabler, eller rekkefølgen på lenking av programfiler. Den egentlige årsaken til endringer i ytelse viser seg å være forskjeller i organiseringen av programkode og data minne. Gjennom eksperimenter og nøyaktig måling ved bruk av "performance counters", kan vi identifisere konkrete arkitekturspesifikke egenskaper ved prosessoren som fører til bias. Vi ser spesifikt på Intel Core i7 "Ivy Bridge"-arkitekturen. Begrensninger i "Loop Stream Detector", samt en effekt kjent som "4K aliasing" studeres i detalj. Vi viser at bias ikke nødvendigvis er uforutsigbart, og illustrerer metoder for å unngå dårlig ytelse som følge av slike effekter.

Undersøkelser av programvare for Fourier-transformasjoner og lineær algebra, viser at særlig alias-effekter ofte kan gi signifikante ytelsesforskjeller i praksis. Både FFTW og ATLAS viser seg å være sårbare mot bias. For eksempel kan riktig plassering av data i minne gi over 30 % forbedring av allerede høyt optimaliserte rutiner for matrise-vektor-multiplikasjon. Å ta høyde for plassering av kode og data i minnet er viktig, særlig for utviklere og brukere av programvare hvor ytelse står sentralt.

Acknowledgements

I would like to thank my supervisor Anne C. Elster for her support during this past year. A special thanks to my co-supervisor Rune E. Jensen, whose continuous assistance and expertise on performance counters helped me navigate through this challenging topic.

The HPC-lab at NTNU provided the hardware and resources necessary to complete this work, for which I am grateful. I would also like to thank the other master students there, for providing a great work environment and invaluable feedback.

Contents

Chapter 1

Introduction

Optimizing for performance is an important topic in computer science, and high performance computing (HPC) in particular. A great amount of research and effort is put into designing better algorithms and compiler optimizations, in order to produce an "optimal" executable. However, performance of computer programs is not just a function of the collection of machine instructions executed. Various external properties, such as the operating system, or even room temperature, can have have an impact on how fast programs execute: Modern microchips can be sensitive to temperature, possibly operating on higher clock rates in cold environments. Characteristics of the operating system can decide things like the placement of code and data in memory, which in turn interacts with various hardware mechanisms. Corner cases in hardware introduces potential *performance cliffs*; If for example a memory access happens to cross a page boundary, the cost of an additional TLB miss can be significant. The set of conditions under which a program executes is called the *execution context*, which includes virtual memory layout in particular. Variations within contexts can lead to programs being *biased* towards certain configurations. Measuring performance of the same program on two machines with identical hardware can sometimes yield very different results, an effect known as *measurement bias*.

Previous work has studied parameters such as the size of Unix environment variables and program link order [18, 17]. Both are found to potentially have a significant effect on performance, even in standardized benchmarks. Unfortunately, the effects also appear to be unpredictable, and therefore difficult to deal with. This poses a problem for researchers and performance analysts, who will need to account for effects of measurement bias with more rigorous methodologies and statistical methods.

In this thesis, we will try to unveil the actual causes of bias that can be experienced by altering memory layout. Both the size of environment variables and link ordering ultimately has an effect on memory layout of running processes. We will *not* look at things like cache efficiency, which often can be the reason for bad performance under certain memory contexts. Instead, we look at effects from less known optimizations in the out-of-order execution engine and instructions fetch pipeline. Because bias effects are intrinsically connected to various hardware features, we will focus on the Intel® Core™ i7 "Ivy Bridge" architecture specifically. The goal is to identify specific hardware features that causes measurement bias, and model how they interact with software programs. With a better understanding of intricate properties of the CPU, we show how one is able to not only predict, but also explicitly optimize for beneficial memory layouts.

1.1 Motivation

In the paper appropriately named "Producing Wrong Data Without Doing Anything Obviously Wrong" by T. Mytkowicz et al. [17], the authors show how simple changes to the execution environment can have an impact on performance. One of the parameters they studied was the size of Unix environment variables. Environment variables contain various information about the system, for instance the name of the currently logged in user, home- and current directory path. It seems unlikely that the contents of these variables should have any significance on program performance, yet previous work show that the effects can be enormous.

```
static int i, j, k;

int main() {
    int g = 0, inc = 1;
    for (;g<65536;g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
```

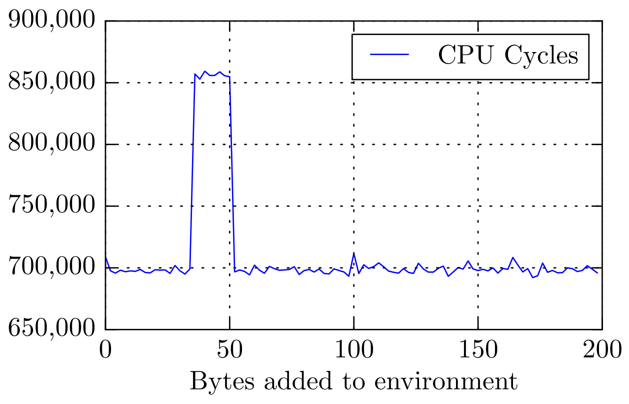

Figure 1.1: A simple C program with significant performance variations under different Unix environment sizes. The example is adapted from [17], with performance measurements updated for the Ivy Bridge processor.

This article is particularly interesting because of an intriguing little C program, which is reproduced here in Figure 1.1. The authors observed that this program was noticeably affected by changes to the environment variables, performing much worse (in terms of cycle count) for some environment sizes. Interestingly, we are able to reproduce very similar results for our newer Ivy Bridge architecture. Plotting the number of cycles executed for different environment sizes, we see that the cycle count suddenly increases by more than 20 %. The length of your name – by extension you user login – could in principle be the tipping point between good and bad performance when running this program.

What is the "correct" number of cycles here? One could say that in most cases, it should be around 700,000, although on average it is somewhat higher. On a machine with just the right environment size, one might declare that the cycle count is about 850,000. This illustrates how external properties can bias performance towards certain environments. Previous work focuses mostly on how to mitigate effects of bias, and avoid drawing the wrong conclusions based on misleading measurements [16, 17]. Our goal is to gain a better understanding of exactly what mechanisms of the processor causes these effects. With more accurate models of how bias occur, we will be able to actively avoid these "spikes" in performance, and gain a real speedup on average.

1.2 Outline

The remaining parts of this thesis structured as follows:

Chapter 2 presents some of the previous work done on measurement bias, showing its severity as well as proposed solutions to mitigate or compensate for it. A brief overview to the Ivy Bridge microarchitecture and use of hardware performance monitoring is given as background material. We also introduce important concepts in virtual memory layout on 64 bit Linux systems, explaining the memory execution context.

Chapter 3 discusses the experimental setup and methodology used in the next chapters.

Chapter 4 investigates several potential causes of bias, showing how certain hardware features are sensitive to changes in virtual memory layout. Specifically, we show how 4K address aliasing and the Loop Stream Detector can explain effects observed by increasing environment size or changing link order.

Chapter 5 applies knowledge of bias effects from the previous chapter, looking at how they apply to real world applications. We take an in-depth look at FFTW, showing how even this highly optimized Fourier transform library can be affected by memory layout. We also show how to get significant performance gains for BLAS matrix-vector routines, by actively avoiding address aliasing of heap-allocated memory.

Chapter 6 summarizes results from previous chapters, and provides directions for future work.

The appendices contain supplementary and reference material:

Appendix A lists the subset of the available performance counters on Ivy Bridge that are relevant to our discussion.

Appendix B lists the script that is used for collecting performance metrics under varying environments.

Chapter 2

Background and Related Work

This chapter presents some of the previous work done on measurement bias in performance analysis. A brief overview of the Intel Core “Ivy Bridge” architecture is included, introducing performance counters, and highlighting some of the features and hardware optimizations that can cause performance cliffs. An introduction to virtual memory, linking and loading on 64 bit Linux systems is provided as background material – explaining important factors of the program execution environment.

2.1 Observer Effect and Measurement Bias

Several papers dealing with bias in performance analysis have been published. Mytkowicz et. al. provide an excellent introduction to measurement bias, together with the closely related phenomenon of observer effect, in the paper “Observer Effect and Measurement Bias in Performance Analysis” [18], and “We have it easy, but do we have it right?” [16]. The authors draw parallels to the social and natural sciences, scientific fields where considerable care is taken to avoid observer effect and bias in experimental setups. They argue that the methodology currently employed in analysis of software is lacking. Because observer effect and bias in experiments is often ignored (or improperly accounted for), performance analysis suffers from “poor-quality data”. In literature surveys, they find that very few authors considers these effects when evaluating results.

Observer effect

Observer effect occurs when the act of observing something changes its characteristics. In performance analysis, we are interested in observing a variety of properties; A useful property for evaluating a database application might be the number of transactions per second, while the cycle count or number of cache misses might be more relevant when implementing a hash function. Many low level metrics, such as the number of CPU cycles used, can be acquired (or observed) using hardware performance counters. In other cases, we need to add instrumentation to the program being measured, for example in the form of incrementing counter variables for every function call. Mytkowicz et al. [18] show that both strict use of hardware counters, as well as adding software instrumentation, is vulnerable to observer effects. Despite using techniques with minimal intrusion and overhead to the original program, observer effects can sometimes significantly skew performance measurements.

Measurement bias

The performance characteristics of software changes not only from observations, but also from external properties of the experimental setup. Two, seemingly innocuous, properties of the execution environment are discussed:

- Link order of object files: The order of which the .o files are provided to the linker. Usually more than one valid permutations exist, and different orderings produce functionally identical programs.
- Shell state, specifically environment variables. The contents of environment variables on a given system depends on many factors, such as the user currently logged in (stored as "USER" in Linux), or which programs are currently installed, adding directories to the "PATH" variable.

The authors study how program performance for different compiler optimization levels is affected by adding characters to environment variables or changing program link order. Their results show that standardized SPEC benchmarks are sensitive to both of these properties. Different conclusions about O2's efficiency versus O3 can be drawn depending on the environment.

There are many other properties one could consider as possible sources of bias. With link ordering causing performance variations, one must also consider the different *versions* of compilers or linkers used. Different compiler versions might generate different code – resulting in variations in memory layout of code and data. In general, the exact implementation of any software system that interacts with program execution must be considered, including compiler toolchain and operating system used.

2.1.1 Causes of Measurement Bias

In a following paper by Mytkowicz et al. [17], the effects previously presented are reiterated, along with a more detailed analysis of measurement bias. In addition to pointing out sensitivity in large benchmark programs from the SPEC suite, the authors also present a small, isolated C program with very high sensitivity to change in environment size. We discuss this particular program in the introduction chapter, showing that similar bias effects appear on newer architectures as well.

For both changing environment variables and link order, the authors point to changes to the virtual memory layout being the real cause of performance variations.

- Changing environment variables has an effect on where the stack is placed in memory at runtime. A copy of the environment is loaded into virtual address space before the call stack starts, thus increasing the environment size offsets the initial stack address. The authors suggest that altering the addresses of stack-allocated variables at runtime can have an impact on things like alignment in cache.
- Changing link order can change the virtual addresses of instructions. Using hardware simulators, the authors show that performance also depends on code alignment in memory. For instance, when a hot loop fits entirely in a single cache line, the number of accesses to instruction cache can be reduced. The authors speculate that the “Loop Stream Detector” might cause bias on the Intel® Core™ 2, a hardware optimization that is supposed to speed up instruction fetching of hot loops.

Despite demystifying the causes of bias to some degree, this paper provides no satisfactory explanation of exactly how or what hardware mechanisms causes bias.

2.1.2 Dealing with Bias

Previous work conclude that measurement bias is *unpredictable*, and difficult to avoid in experimental setups. In literature surveys, Mytkowicz et al. find that almost no papers account for measurement bias [18, 17]. This is problematic, because conclusions reached from running biased experiments can easily be misleading, or even outright wrong. Measures have to be made to ensure the results obtained by performance analysis are valid. The authors provide some guidelines for how to conduct sound experiments:

- Diversify benchmarks, using statistical methods over a set of diverse benchmarks will help cancel out bias. Their study of the SPEC suite shows that currently widely used benchmark suites are not diverse enough.
- Randomize experimental setup, performing multiple measurements under different configurations of variables that are known to cause bias.

2.2 Exploiting Bias for Optimization

So far we have pointed out the problems bias causes for performance analysis. A different perspective on these effects is to look at it as a potential for optimization.

2.2.1 Blind Optimization

An optimizing compiler needs to have some sort of abstract model of the target processor or machine, in order to emit an “optimal” set of instructions. As CPUs become more and more sophisticated, creating a good model for optimization is difficult. Additionally, with bias effects deemed *unpredictable*, creating an accurate model becomes impossible. In "Blind Optimization for Exploiting Hardware Features" by D. Knights et al. [14], the authors take a different approach to optimization, disregarding the model altogether. Using automatically generated *program variants*, finding an optimal program can be reduced to a searching problem within a *variant space*. The authors consider instruction alignment of functions and global variables as the variant space. New program variants are generated by inserting alignment directives in the assembly, changing the memory addresses of each function or variable independently. A subset of program variants are generated, and evaluated by measuring execution time. Using blind optimization, the authors are able to achieve up to 12.6 % speedup on some SPEC benchmarks, compared to compiler optimized code.

A related concept to blind optimization is *feedback directed optimization* [22]. The idea behind this technique is for compilers to use empirical data to determine the optimal optimization parameters to apply. Parameters are determined through multiple iterations of compiling and profiling. This is useful for instance in determining the number of loop iterations to unroll, different register allocations, and code layout.

2.2.2 Assembly-Level Optimizations

Recognizing that current hardware models are lacking, Hundt et al. [8] presents MAO, an "Extensible Micro-Architectural Optimizer". This is an assembly-to-assembly translator, containing architecture-specific rules to optimize machine code emitted by the compiler for x86 and x86-64 architectures. Many rules attempt to fix "sloppy" code generated by GCC (or other compilers), for instance redundant `test` instructions. The following example can be shortened by removing the last instruction, as condition flags are set implicitly by the `sub` instruction.

```
sub   16, %rax
test  %rax, %rax
```

Many more of these rules are encoded in MAO, resulting in reduced code size and use of more efficient instructions.

In addition to optimizing for the ISA, the code is also tuned for specific properties of Intel microarchitectures. As an example, MAO considers instruction address alignment for hot loops, inserting additional alignment directives if necessary. By aligning frequently executed code segments to 16 byte boundaries, the instructions are more likely to fit in decode "chunks", which are 16 B on the Intel Core 2. In previous work on measurement bias by Mytkowicz et al, the Loop Stream Detector is suggested as a potential cause of bias. The authors identifies that this hardware optimization depends on 16 byte instruction alignment on the Core 2 architecture.

MAO uses this knowledge to align code to fit in as few decode chunks as possible, allowing the LSD to kick in and avoid instruction fetch for some hot loops. Another interesting architecture specific optimization considers available execution ports in the Core 2 when scheduling instructions.

Inspired by the blind optimization technique introduced by Knights et al. [14], MAO also attempts to insert `nop` instructions at random. This has the effect of moving surrounding code around, possibly hitting some optimal configuration not modeled or caught by any of the other optimization rules.

This paper is particularly interesting, because it shows that modeling complex hardware mechanisms is in fact feasible. By encoding highly architecture-specific knowledge on things like optimal instruction alignment, the authors are able to improve on compiler-optimized code.

2.3 Hardware and Performance Monitoring

Bias effects generally depend on specific processor features, and "performance cliffs" caused by them. A rudimentary understanding of the underlying hardware is necessary in order to understand why performance varies during different execution contexts. We limit our study to the Intel Core "Ivy Bridge", currently the most recent Intel architecture. Of course, a comprehensive introduction to the inner working of any processor is way beyond the scope of this thesis. Instead, we will provide a brief overview of some architectural features, and in particular those that might cause bias due to differences in virtual memory layout. For a general introduction to concepts in computer architecture, refer to [19].

To gain accurate measurements and diagnostics, we will use hardware performance counters. Intel architectures have extensive monitoring capabilities, and we give a brief overview of how this can be utilized.

2.3.1 "Ivy Bridge" Microarchitecture

The name "Ivy Bridge" is a code name for what is also referred to as the "3rd Generation Intel Core Microarchitecture", or alternatively "Intel Core Microarchitecture Code Name Ivy Bridge". These are the terms that are used in the manuals, but we will refer to architectures mostly by code name throughout the rest of the thesis. Preceding Ivy Bridge is "2nd Generation Intel Core Microarchitecture" with code name "Sandy Bridge". Many of the architectural features found in Ivy Bridge can also be found in Sandy Bridge and earlier Core architectures.

For our purposes, a general overview and idea of what happens during program execution is sufficient. Figure 2.1 shows a high level view of the processor pipeline, consisting of a front end, out of order engine, execution core, and cache hierarchy. We will give a brief overview of each component, while a comprehensive reference can be found in the vendor manuals [11, 10].

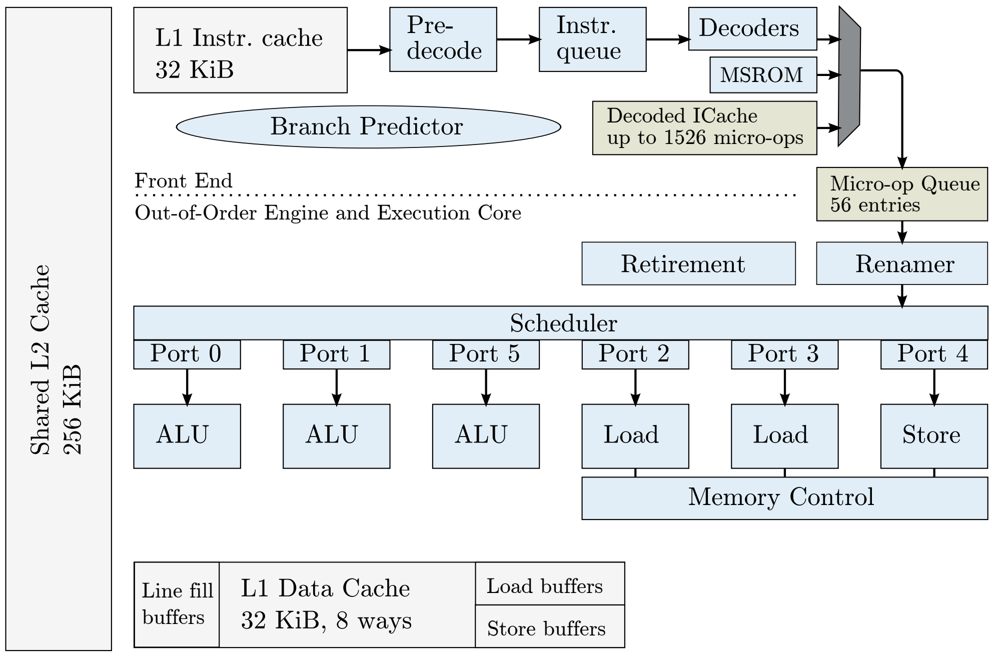

Figure 2.1: Ivy Bridge pipeline, adapted from [10]

Front End

The front end is responsible for fetching and decoding instructions, feeding the execution pipeline with a constant stream of micro-operations. The branch predictor is used to fetch instructions speculatively, along the most likely path of execution.

There is an important distinction between IA-32 (or Intel 64) instructions[1] and *micro-operations* ("micro-ops"): Assembly instructions emitted by a compiler are decoded into micro-operations used internally in the processor. The x86 ISA is a *complex* instruction set computer (CISC) architecture, as opposed to a *reduced* instruction set architecture (RISC). Complex instruction set architectures support more sophisticated operations, such as combinations of memory accesses and arithmetic operations in a single instruction [19]. For historical reasons and sake of backwards compatibility, the x86 instruction set has been constantly expanded with new functionality. To manage the complexity of a CISC ISA, the processor reduces complicated instructions to much simpler micro-operations. The micro-operations allow the processor to be organized more like a RISC architecture internally.

Decoding assembly instructions is a costly operation in hardware, and a lot of logic in the front end is dedicated to avoid this stage as much as possible. The Decoded ICache is a cache holding recently executed micro-operations, containing up to 1526 entries. When the next block of code to execute is determined by the

[1]Intel uses IA-32 and Intel 64 for what is also commonly known as x86 and x86-64 instruction set architectures. We will use these terms interchangeably.

branch predictor, the micro-op cache is searched first. Intel claims that typical hit rates exceed 80 %, and approaching 100 % for hot spots [10]. If not found in the micro-op cache, the "legacy" fetch-decode pipeline is used. New blocks of instruction memory goes through several decode stages, before finally being ready to execute as micro-ops. To fill "holes" in the instruction stream, a queue of micro-ops sits between the front end and the rest of the pipeline.

Macro-Fusion To increase the number of instructions that can be executed each cycle, pairs of instructions can sometimes be fused by the front-end into one equivalent, more complex, operation. With macro-fusion, certain pairs of assembly instructions can be combined to a single micro-operation. This is restricted to pairs where the first instruction modifies condition flags (such as `test` or `cmp`), and the other is a conditional jump. Pairs like these appear often in practice, improving the instructions per cycle metric when macro fusion is applied.

Loop Stream Detector The Loop Stream Detector is, with some restrictions, able to detect software loops in the micro-op stream. It analyzes micro-ops residing in the Micro-op Queue, and identifies chains of repeatedly executed operations. When a loop is detected, the LSD locks the micro-ops in the queue, disabling any further ICache lookup as well as the whole decode pipeline. Micro-ops are streamed directly from the queue, until a branch mispredict causes normal operation to resume. This optimization can save power by disabling logic, but also give a performance gain in cases where the front end is a bottleneck.

Out-of-Order Engine and Execution Core

One of the key features of modern processors is instruction level parallelism. A great amount of logic is dedicated to be able to issue multiple instructions per cycle, increasing the throughput. Micro-ops are delivered sequentially and in-order from the front end. The out-of-order engine then views the stream of micro-ops as a *dataflow* problem. A window of available micro-ops are analyzed for potential data dependencies, and issued for execution accordingly. Multiple operations can be issued simultaneously, as long as there are no dependency violations. There are six execution ports, giving a maximum of six micro-operations issued by the scheduler each cycle.

Cache Hierarchy

Each processor core has an L1 data and instruction cache of 32 KiB[2] each, and a unified L2 cache of 256 KiB holding both data and instructions. The Line Fill Buffer

[2]To avoid any confusion with SI units, we will use KiB (kibibyte), MiB etc for numbers with base 2. 1 KiB is 1024 bytes, 1 MiB is 1024 KiB, and so on.

(LFB) is a component of the L1 data cache, serving as a buffer for pending loads between L1 and L2. An L3 cache of 8 MiB, also known as last level cache (LLC), is shared among all cores via a ring connection.

Prefetching Cache lines are loaded speculatively by *prefetching*, based on a prediction of what data the program will request in the future. There are several prefetchers, operating between different cache levels. Prefetching is particularly helpful for streaming applications, with linear data accesses. With prefetching, the next cache lines can be loaded before the program requests them, improving performance by masking memory latency. There is also a software mechanism for prefetching, allowing programmers to give hints to what data should be preloaded into cache.

Store Forwarding If a load follows a store to the same memory location, the store value can be forwarded to the load operation directly. Store forwarding avoid accesses to data cache, allowing loads to execute faster.

Memory Disambiguation In cases where a store operation is followed by load, there is a dependency if both instructions refer to the same memory address. The processor can not safely issue the load operation before the store (out of order) until the addresses are calculated, and determined to not overlap. A conservative approach is to block all load instructions until previous store addresses are resolved. With *memory disambiguation*, the hardware makes a prediction about whether a conflict will occur, scheduling loads speculatively [6, 26]. Loads that are predicted not to have any dependencies are allowed to read data from L1 cache, before previous store addresses are resolved. The predictions are later verified before retirement, re-executing instructions that were affected by actual conflicts.

Load and store buffers hold the values of speculatively executed memory operations, sometimes collectively referred to as Memory Order Buffer.

2.3.2 Performance Counters

Modern CPUs have build-in support for performance monitoring in hardware. Performance counters are special-purpose registers that can be configured to *count* various hardware events. The processor contains a dedicated Performance Monitoring Unit (PMU), which can be programmed to count things like cycle count, number of branch misses, or retired micro-operations. Intel CPUs have had this functionality for many generations. Performance counter metrics are commonly used for detailed tuning and profiling of software. There are many benefits of using hardware support for performance monitoring, including accuracy and minimal overhead. Problems with observer effects and bias are avoided, because performance monitoring does not interfere with software.

Table 2.1: Examples of performance counters available on Ivy Bridge. The official reference can be found in [12], and a collection of the most relevant events for this thesis is included in Appendix A.

Event	**Umask**	**Mnemonic**	**Description**
0x3C	0x00	CPU_CLK_UNHALTED. THREAD_P	Counts the number of thread cycles while the thread is not in a halt state
0xC0	0x00	INST_RETIRED.ANY_P	Number of instructions at retirement
0xA2	0x01	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason

Performance events are identified by two numbers, called *event code* and *unit mask*. Events are also referred to by names, or *mnemonics*. A few examples are listed in Table 2.1, including CPU_CLK_UNHALTED.THREAD_P counting number of cycles, and INST_RETIRED.ANY_P for counting number of dynamic instructions executed. About 200 different events are available on Ivy Bridge [12].

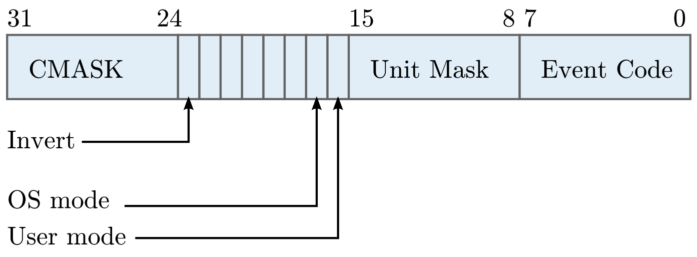

Figure 2.2: Performance event select register. Adapted from [12]

Counters are configured up by writing to 32 bit *event select* registers. Figure 2.2 illustrates the bit field for initializing counters on Ivy Bridge.

- Unit mask and event code are written to the lower two least significant bytes. As an example, looking at Table 2.1 we find the values for INSTR_RETIRED. ANY_P to be 0x00 and 0xC0 respectively.
- The remaining bits can be toggled to impose various constraints on what is counted. We will typically toggle OS mode and user mode such that statistics is reported from code executing in user mode only. Counters are updated each cycle, adding the number of events occurring to a cumulative value. An *invert* bit can be set to count cycles where an event does *not* occur.
- On each cycle, the event count is compared to the value specified for counter mask (CMASK). The performance counter is not incremented if the value is

below what is specified for counter mask. For instance, counting INSTR_RETIRED.ANY_P with a counter mask of 2, will count the number of cycles where at least two instructions are retired.

On Ivy Bridge, there are three *fixed* and eight *general-purpose* counter registers, for a maximum of 11 performance events monitored simultaneously on each core. The fixed counters are set to count retired instructions, CPU cycles and reference cycles. With some restrictions, the remaining general-purpose registers can be programmed to count arbitrary events. The official documentation of available performance counters and their usage can be found in Volume 3B of the Software Developer's Manual [12]. A shorter reference is included in Appendix A, listing only counters that are relevant to the discussion throughout the remainder of this thesis.

2.3.3 Using `perf`

There are multiple ways of accessing the processor's performance monitoring facilities from software. Support is needed from the operating system, as kernel mode privileges are needed to manage the low level PMU hardware. Linux systems provide a kernel interface called *perf_events*, which can be used along with the perf[3] tool. Perf is a utility program for interfacing performance counters from user code on Linux. It is relatively easy to use, with support for many performance counters by mnemonics such as "cycles", "instructions", and "branch-misses". We will mostly use the `stat` command, which takes a list of performance counters and a program to benchmark as arguments. As an example, the following command reports the total cycle count from executing ls (listing files and directories).

```
perf stat -e cycles ls
```

Not all counters are available as mnemonics. Arbitrary performance events can be used by specifying their respective unit mask and event code, available in the documentation [12]. Perf uses the event select bit-format fairly explicitly, meaning we can provide a hexadecimal number corresponding to the desired register value to specify which counter to use. The perf code for RESOURCE_STALLS.ANY is "r01A2", with 0x01 and 0xA2 for unit mask and event code respectively. An initial 'r' character signifies that a "raw" counter value is specified. The user mode bit can be toggled by appending a 'u' character, as in `event:u`.

```
perf stat -e r01a2:u ls
```

Simply concatenating unit mask and event code and appending a trailing `:u` is what we want to do in most cases.

[3]`perf`: Linux profiling with performance counters, https://perf.wiki.kernel.org

2.4 The Execution Context

In order to understand why bias effects occur, it is necessary to look at some of the low-level aspects of how processes and memory is managed by the operating system. In particular, we will see that changes to memory layout in virtual address space is an important factor. Some knowledge of ELF executables and virtual memory is necessary for the remainder of the thesis. We will provide a brief overview of how code and data is mapped to virtual memory on 64 bit Linux systems. For a more comprehensive reference to topics in operating systems, refer to [23].

2.4.1 Virtual Memory Layout

Virtual memory is an abstraction over the physical memory (RAM), allowing every program operate within its own isolated address space. Virtual addresses are organized into *pages* of 4 KiB, and a *page table* managed by the operating system defines a mapping to *frames* in physical memory [23].

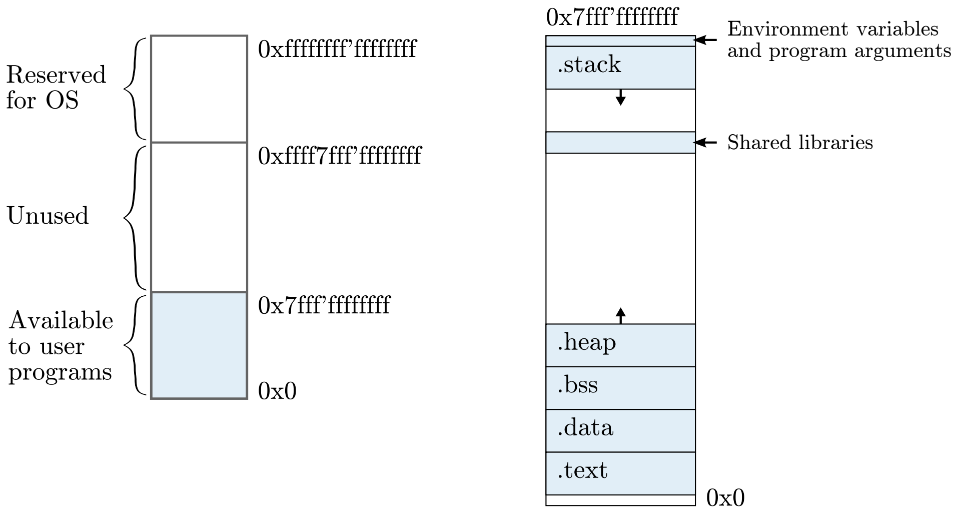

Figure 2.3: Organization of virtual memory of a running process on a typical 64 bit Linux system

The memory image of a running process in virtual address space is illustrated in Figure 2.3. Code and data of executable programs are divided into several sections, the most important being stack, heap, bss, data and text.

stack Stack frames, containing function parameters and local variables. Located somewhere around the top (address 0x7fff'ffffffff) and expanding downwards.

heap Dynamically allocated area. Located above the uninitialized data segment (bss), and growing upwards as more space is required. Calls to `malloc` and related functions can be used to request heap memory from the operating system.

bss Uninitialized data, containing program variables with no value specified. This area is implicitly initialized to zero once the program is loaded.

data Initialized data, containing program variables that have some assigned value at compile time.

text Executable binary code, located at the lower end of the virtual address space.

On 64-bit machines, addresses range from 0 to 2^{64}. Note that effectively only 48 out of 64 available bits are used for addressing. The lower 0x0 through 0x7fff'ffffffff is addressable to user programs, while the upper segment of addresses from 0xffff8000' 00000000 through 0xffffffff'ffffffff is reserved for the operating system. This leaves a large gap of addresses that are not used, as current hardware does not actually support addressing the whole 64 bit range [15].

Environment Variables

As discussed in previous work on measurement bias [17], the size of Unix environment variables can affect the placement of stack. Environment variables are allocated above the stack, pushing the initial stack address downwards. Because of this, the position of stack can be manipulated by manually changing environment variables. Note that program arguments are also located above the stack, thus providing different arguments can potentially lead to similar effects.

Executable File Format

Modern Unix systems represent executable program files in the Executable and Linkable Format (ELF) [2]. The same format also handles object files (.o) and shared libraries (.so). The different steps in a typical compilation process leading up to an ELF executable is shown in Figure 2.4. One or more source files (for example written in C) are first translated to assembly code by a compiler. An assembler translates the assembly code into machine code, wrapped in an ELF file format. Multiple object files can be linked together, forming a single executable file. The linker resolves any symbols and dependencies between the object files.

ELF files are structured in a set of sections, such as ".text", ".data" and ".symtab". Some sections are *allocable*, meaning their contents are copied into virtual memory before the program is executed. An example of an allocable section is the text area, which contains all the executable machine code. The data section is also allocable, typically storing things like constants and initialized static variables. Virtual addresses of code or data stored in allocable section are determined statically, and can be found by reading the contents of the ELF file. The following example shows disassembly of the first few instruction in the main function of a program, which in

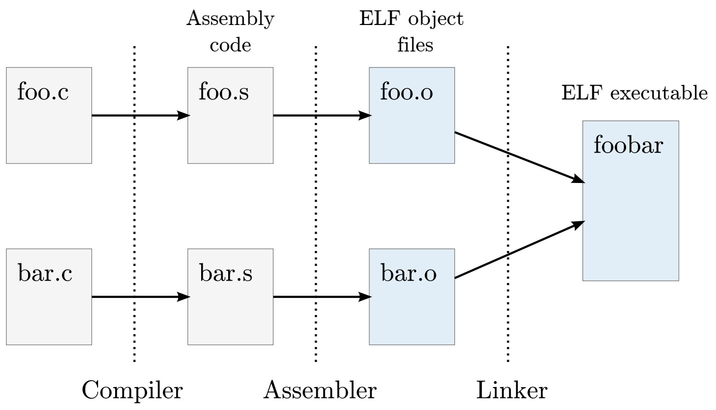

Figure 2.4: Overview of important steps of the compilation process

this case starts at address 0x400544 [4].

```
0000000000400544 <main>:
  400544:       55                      push   %rbp
  400545:       48 89 e5                mov    %rsp,%rbp
  400548:       48 83 ec 10             sub    $0x10,%rsp
  40054c:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400553:       bf 04 00 00 00          mov    $0x4,%edi
```

Before a program can be executed, it needs to be loaded into virtual memory in some way. The ELF executable file includes the path to an interpreter, a program that is responsible for loading code and data, setting up a stack, and loading any dependencies. Files compiled for execution on a 64 bit Linux platform will typically point to a loader in /lib64/ld-linux-x86-64.so.2. This file can be a symbolic link to the actual program. The loader maps each allocable section of the object file into virtual address space, such as the text and data areas. Stack, heap and bss areas are initialized, resulting in a process image like the one shown in Figure 2.3. Dynamically linked and shared libraries are mapped into virtual memory as well, appearing somewhere between the stack and heap segments.

Address Space Layout Randomization (ASLR)

The process of mapping sections to virtual memory is not necessarily deterministic. If addresses can be known a priori, programs will be more vulnerable to buffer overflow attacks [21]. If an attacker manages to inject malicious code, he can use the known virtual addresses to access data or call specific functions. One technique known as “return to libc” attacks uses this knowledge to call functions in the C standard library, which will be loaded in most scenarios. If dynamic libraries are

[4]Output is generated using the objdump utility, part of GNU Binary Utilities. readelf is another useful program for analyzing ELF files.

always mapped to the same range of virtual addresses, locations of functions in libc can be easily determined.

Address Space Layout Randomization is a technique employed by the operating system to make it more difficult to compromise programs. By default, the placement of stack, heap and location of dynamically linked libraries will vary between each run of a program. For a controlled execution environment, we will need to manually disable address randomization in many cases. This can be done by overwriting the “randomize_va_space” system property.

```
$ sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

The default value is 2, indicating full randomization[5]. To disable heap randomization only, a value of 1 can be written instead. We will use zero to disable all randomization, making virtual memory layout completely deterministic between each run.

2.4.2 Memory Context of C Programs

The C programming language will be used for code examples throughout the remainder of the thesis, as it is both widely known and provides only a thin layer of abstraction over virtual memory. We use the program in Listing 2.1 to illustrate how variables in C programs map to different memory segments. The program contains four variables, utilizing both initialized and uninitialized data, dynamically allocated data on heap, and stack allocated variables.

- Variable `c` and `d` are stack-allocated *automatic* variables [13]. Their run-time addresses depend on where the stack segment is initialized in virtual memory, and can in general not be known ahead of time.

- The address returned by malloc and stored in variable `d`, points to a location on the heap. The exact address depends on the implementation of malloc, as well as the initial position of the heap segment, and is also impossible to predict in general.

- Global variables `a` and `b` should be placed in bss and data segments respectively. Variable `a` is not explicitly initialized, thus it is not necessary to store any value in the executable file. The C standard specifies that static variables are guaranteed to be initialized to zero [13], which will happen by default if allocated in the bss segment. The value 42 is allocated for variable `b` in the data section of the executable. Virtual addresses of static variables are determined at compile time, and can be found in the ELF object file’s symbol table [2].

[5]A description of the different values can be found in the documentation for sysctl, https://www.kernel.org/doc/Documentation/sysctl/kernel.txt

Listing 2.1 Simple C program with data allocated in four different segments of virtual memory

```
#include <stdlib.h>
#include <stdio.h>

static int a;       /* .bss */
static int b = 42;  /* .data */

int main() {
    int c = 0;      /* .stack */
    int *d = malloc(sizeof(int)); /* .heap */
    printf("stack: %p heap: %p bss: %p data: %p \n", &c, d, &
       a, &b);
    return 0;
}
```

Executing the same program multiple times with ASLR enabled will yield different output for stack and heap addresses. Each variation constitutes a different execution context, with potentially different performance characteristics. Below is an example output.

```
stack: 0x7fffffffe1ec heap: 0x602010 bss: 0x601040 data: 0x601028
```

Chapter 3

Methodology and Experimental Setup

This chapter presents the test setup and experimental methodology used to identify sources of bias in Chapter 4, and case studies in Chapter 5. When studying bias effects, carefully controlling all relevant properties of the environment is crucial. We first describe how our experimental setup is configured with respect to operating system settings, and use of “best practices” in performance analysis. Methods used for acquiring and analyzing performance counter measurements is also described.

3.1 Setup and Configuration

Measurement bias is by definition a product of variations in the execution environment. To produce reliable results, we need to properly control every variable that can affect measurements. Previous work on measurement bias and observer effect describes a set of best practices for how the measurement infrastructure should be configured [18, 17]. Based on this, we make the following configuration to our experimental setup:

- Unless specified otherwise, address space layout randomization (ASLR) is kept disabled. This is necessary when testing effects from changing memory context, and often required to make results reproducible.

- Hyper threading is disabled, ensuring that only one thread runs simultaneously on each core. Two threads competing on hardware resources is another potential source of bias, which we will not be studying.

- System load is kept at a minimum to avoid interference with other processes and tasks.

- Automatic CPU frequency scaling is disabled, keeping the clock rate constant.

Table 3.1: Experimental setup

Processor	Intel® Core™ i7-3770 @ 3.40 GHz Family: 0x06, Model: 0x3A
Memory	16 GB @ 1333 MHz
Operating System	64-bit Ubuntu 12.04 LTS (3.2.0-41)
Toolchain	GCC 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)

Hardware and software configuration of our test machine is shown in Table 3.1. It features an Intel Core i7-3770 "Ivy Bridge" processor and 16 GB RAM. The operating system is Ubuntu version 12.04, which at the time of writing is the most recent release with long term support. We use GCC version 4.6.3 as our compiler toolchain.

3.2 Performance Analysis

We use `perf stat` to acquire hardware performance counter statistics, which should introduce minimal overhead and observer effect. In many cases we need to collect a large number of counters for the same program, while there is a limit to the number of performance counters collected simultaneously (see Chapter 2). We use a Python script to encapsulate repeated invocations of `perf stat`. Being only a thin wrapper around invocations of perf, this script does not affect program execution or introduce any bias by itself. Measurements are aggregated over several runs, enabling us to collect all of the about 200 performance counters available on our architecture [12]. The implementation can be found in Appendix B. A reference to performance counters discussed in the remaining chapters can be found in Appendix A.

Environment Size Previous work points to changes in environment variable size as a cause of bias, and we will study this effect in detail. Unless otherwise specified, we use a minimal environment for all our benchmarks[1]. A script is used for collecting performance counters under varying environment sizes. To measure statistics with n bytes added to the environment, we set a dummy variable to 0^n (repeated zero characters n times).

Correlation Many experiments consists of collecting a large number of performance counter statistics over a series of different execution contexts, for example incrementing environment variable size. To filter out events that are likely to explain any bias effects, we will use the *Pearson correlation coefficient* [25]. The correlation

[1]A completely empty environment is not possible, because `perf stat` itself adds some variables to the environment.

coefficient is a numerical value between -1 and 1, which can be used to determine the linear relationship between two data series X and Y. Values close to $+1$ or -1 indicates strong positive or negative correlation, respectively, while values close to 0 indicate no correlation. With multiple series of performance counter measurements, a linear correlation coefficient serves as a crude indication of possible relationships between counter values. To filter out indicators of bias, we will look at correlation between cycle count and other counter values in particular.

Cache Analysis Cache conflicts can in many cases be a potential explanation for bias effects observed by changing memory layout. If some memory configuration causes contention on a particular cache line, then repeated misses can reduce performance. Bias caused by cache issues is not the focus of this thesis. However, cache is generally a *likely* explanation of performance variations. In our experiments, we will carefully monitor various cache metrics in order to rule out cache as the underlying cause of bias. Relevant events include the hit rates of load micro-ops for each level of cache, which can be monitored by the following performance counters [10]:

- MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS
- MEM_LOAD_UOPS_RETIRED.L1_HIT_PS
- MEM_LOAD_UOPS_RETIRED.L2_HIT_PS
- MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS
- MEM_LOAD_UOPS_RETIRED.LLC_MISS_PS

Together, these events account for all cache accesses for load operations within a single core. Every cache access is either a *hit* in Line fill buffer, L1, L2 or L3, or an L3 *miss*. Changes to hit rates for different levels of cache is relevant when evaluating cache performance. Note that it is also possible for load operations to hit cache lines in other processor cores. This is scenario is mostly relevant for multithreaded applications, which we will not study.

Visualization Plots are generated with Python's matplotlib package [9], using the csv exported data from our benchmark script.

3.3 Approach

Our goal is to investigate possible *causes* of measurement bias, and if possible how this can be used for optimization. We have chosen to approach this problem from two angles; first through low-level analysis of smaller examples, and then later with more high-level case studies of real applications.

In Chapter 4, we present results from *experimental* work, trying to identify hardware and software mechanisms that can lead to bias. Smaller synthetic code examples are used, which are easy to analyze. Bias triggers identified in previous work will be used as a starting point, in particular environment size and link order [17]. Extensive measurements using hardware performance counters are performed. Using correlation techniques, we can filter out events that are likely to explain bias. The goal is to develop a solid understanding of the underlying hardware mechanisms that can trigger bias.

Results and experiences from the experimental work is transferred to *case studies* in Chapter 5. With a detailed understanding of architectural features causing bias, we can look for specific effects in larger applications. We choose to focus on already highly optimized numerical libraries. Performance critical applications are more likely to already have considered architecture specific optimizations to account for bias effects. If we can find any additional opportunity for improvement, there will probably be incentive to update or enhance these libraries.

Chapter 4

Sources of Measurement Bias

Modern microprocessors are extremely complex in design and functionality. Some features of recent Intel processors includes several layers of cache to camouflage slow memory, multiple prefetchers, speculative out-of-order execution and branch prediction, just to name a few. Hardware features and optimizations interact with memory layout of program code and data in various ways. In this chapter, we will unveil characteristics about two different architectural features in Ivy Bridge, and show how they can bias performance towards certain memory contexts.

First we will look at an *aliasing* effect between memory addresses of loads and stores, causing false dependencies in the out-of-order execution pipeline. This effect can be triggered for example by changing stack position, and can explain bias from altering environment variables. Secondly, we will look at the *Loop Stream Detector*, which in previous work on measurement bias has been suggested as a possible explanation. We therefore choose to study this particular optimization in detail, and show how measurement bias can occur from changing link order.

4.1 Address Alias Effects (4K Aliasing)

An effect known as "4K aliasing" can occur when the addresses of a store instruction followed by a load instruction differ by a multiple of 4 KiB. Consider the following example, first writing a value to the memory address stored in %rax, before reading from the address stored in %rbx

```
mov %r10, (%rax)     /* store to address in %rax  */
mov (%rbx), %r11     /* load from address in %rbx */
```

If for example %rax is 0x601010 and %rbx is 0x604010, the memory accesses are said to be aliased. The difference between the two addresses is 0x3000, which is a multiple of 4096, or 0x1000 in hexadecimal. The memory system issues load and store operations speculatively and out of order to increase throughput and parallelism. Some analysis is done to determine which operations are safe to issue out of order. In the above example, the processor could very well issue the load before the store, as they refer to different addresses. However, on recent Intel architectures only the last 12 address bits are used to determine if operations refer to the same address [10]. The last twelve bits of 0x601010 is 0x010, or 0000’0001’0000 in binary – the same as for 0x604**010**. Only doing a partial address compare, dependencies between load and store operations are sometimes falsely detected. These events can be counted by the following performance counter:

LD_BLOCKS_PARTIAL.ADDRESS_ALIAS False dependencies in Memory Order Buffer due to partial compare on address.

The Memory Order Buffer refers to a collection of load/store buffers within the L1 data cache. It buffers loads and stores of not yet retired instructions, enabling speculative and out of order execution while ensuring that no dependencies are violated [5, 10]. The optimization manual provides a more concrete explanation of the counter, stating it “Counts the number of loads that have partial address match with preceding stores, causing the load to be reissued” [10]. False dependencies due to address aliasing can have a negative effect on performance due to loads being reissued.

Intel Optimization Guidelines Performance implications of 4K aliasing is discussed to some extent in the optimization manual from Intel. There are also some concrete suggestions for how to deal with it:

> ***User/Source Coding Rule 8. (H impact, ML generality)***
> *Consider using a special memory allocation library with address offset capability to avoid aliasing* [10]

This rule concerns heap allocated memory, which can be vulnerable to address aliasing if the implementation of for example malloc often returns memory with identical 12 bit address suffixes. This rule is classified as high (H) impact and medium/low generality, suggesting that it occurs relatively often, and with significant performance implications. In Section 4.1.2 we will study the effects of aliasing of heap allocated memory. Alternative allocator implementations is explored in Section 4.1.3, identifying dynamic libraries as a potential source of measurement bias.

The second rule concerning address aliasing introduces the concept of *padding* variable declarations.

> ***User/Source Coding Rule 9. (M impact, M generality)***
> *When padding variable declarations to avoid aliasing, the greatest benefit comes from avoiding aliasing on second-level cache lines, suggesting an offset of 128 bytes or more* [10]

Aliasing can in some cases be accounted for in software by explicitly handling memory location of variable declarations. We will use this technique to avoid aliasing for both stack- and heap allocated memory. Notice that Intel considers these as a "User/Source Coding" rules. This suggests that optimizing for address aliasing is something that should be considered not only by compilers, but also by programmers.

Outline We will study the effects of 4K address aliasing through a set of examples and small case studies. The purpose is to show how aliasing can explain measurement bias. In Section 4.1.1 we connect address aliasing with changes to environment size – showing how variations in stack addresses can trigger address collisions. Section 4.1.2 discusses common aliasing issues with heap allocated memory. We choose to focus on a limited set of concrete code examples, illustrating how aliasing can affect real programs. Additionally, we show how to use *padding* or similar techniques to programmatically avoid bias in each case. Finally, in Section 4.1.3 we introduce dynamic libraries as another source of bias. In addition to environment variables, we find that different configurations or versions of memory allocators can cause measurement bias by triggering address aliasing.

4.1.1 Bias from Environment Size

As shown in previous work, changing the Unix environment variables can sometimes significantly affect program performance [17]. It is typically not the content of environment variables that are important, but rather the effect their size has on stack position. Environment variables are allocated before the stack when programs are mapped into virtual memory, essentially offsetting all the following call frames and stack allocated variables.

Listing 4.1 Small C program with bias towards certain environment sizes. Adapted from [17].

```c
static int i, j, k;

int main() {
    int g = 0, inc = 1;
    for (; g < 65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
```

Analysis of Micro Kernel

We begin by showing how address aliasing can explain measurement bias for the micro kernel presented in [17], reproduced here in Listing 4.1. This example is interesting for several reasons:

- The bias effects are significant and easily reproducible
- The example code is simple and easy to analyze
- No satisfactory explanation as to what causes bias was given in the original paper

As outlined in Chapter 3, we use perf and an automated script to collect performance counter statistics over a series of runs, incrementally increasing environment size using a dummy variable. The program is compiled using GCC with no optimization. Note that any optimization would likely disregard most of the function as redundant code, and reduce it to return zero immediately. Collecting the cycle count performance counter, we observe a distinct spike over a period of 16 bytes under increasing environment size. Illustrated in Figure 4.1, the difference is over 20 % between the worst and best case.

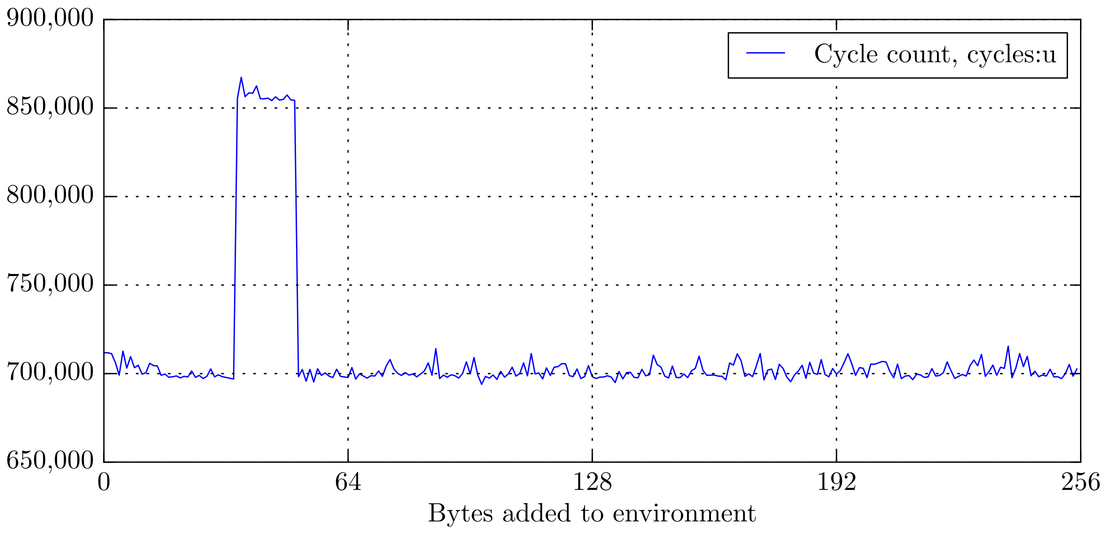

Figure 4.1: Performance variations from offsetting stack position by changing environment size

As a starting point in trying to analyze the potential cause for these effects, we look at data provided by other performance counters. We run the program several times under each environment configuration, collecting all available counters. The most interesting metrics are filtered out by linear correlation coefficient, as described

Table 4.1: Performance counters with more than 0.3 positive or negative linear correlation to cycle count.

Performance Counter	**Perf code**	**Correlation**
UnHalted Core Cycles	cycles:u	1
UnHalted Reference Cycles	bus-cycles:u	0.9999957522
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	r0107:u	0.9941211321
RESOURCE_STALLS.ANY	r01a2:u	0.9922818774
CYCLE_ACTIVITY.CYCLES_LDM_PENDING	r02a3:u	0.9879579406
CPL_CYCLES.RING123	r025c:u	0.9844339013
CYCLE_ACTIVITY.CYCLES_NO_EXECUTE	r04a3:u	0.9200557425
UOPS_DISPATCHED_PORT.PORT_4	r40a1:u	0.7546588979
UOPS_DISPATCHED_PORT.PORT_5	r80a1:u	-0.5346383723
UOPS_DISPATCHED_PORT.PORT_1	r02a1:u	-0.9308766294
UOPS_DISPATCHED_PORT.PORT_0	r01a1:u	-0.9707469127
RESOURCE_STALLS.RS	r04a2:u	-0.986720269

in Chapter 3. The set of counters that most closely follows the cycle count is shown in Table 4.1. Among all the performance counters supported on our architecture, only a few shows significant correlation to cycle count:

- Address aliasing (LD_BLOCKS_PARTIAL.ADDRESS_ALIAS) have a near perfect correlation, indicating that false memory dependencies causes a performance hit.
- Positive correlation with resource stalls supports the aliasing hypothesis. Loads blocked by preceding stores due to false dependence is likely generating stalls. RESOURCE_STALLS.ANY counts stalled cycles from "resource related reasons" [12]. CYCLE_ACTIVITY.CYCLES_LDM_PENDING is a related metric, counting the number of cycles with pending memory loads.
- The remaining counters appears to be less interesting. CPL_CYCLES.RING123 and bus cycles will naturally correlate with cycle count. There are differences in the way micro-operations are scheduled between dispatch ports, which seems more like a by-product of the other events.

Figure 4.2 shows a plot of resource stalls, pending loads and alias, overlaid on cycle count. We see that LD_BLOCKS_PARTIAL.ADDRESS_ALIAS reports zero almost everywhere, and spikes in perfect correlation with cycle count. A higher amount of resource stalls and pending loads seems like a natural consequence of stalling due to aliasing.

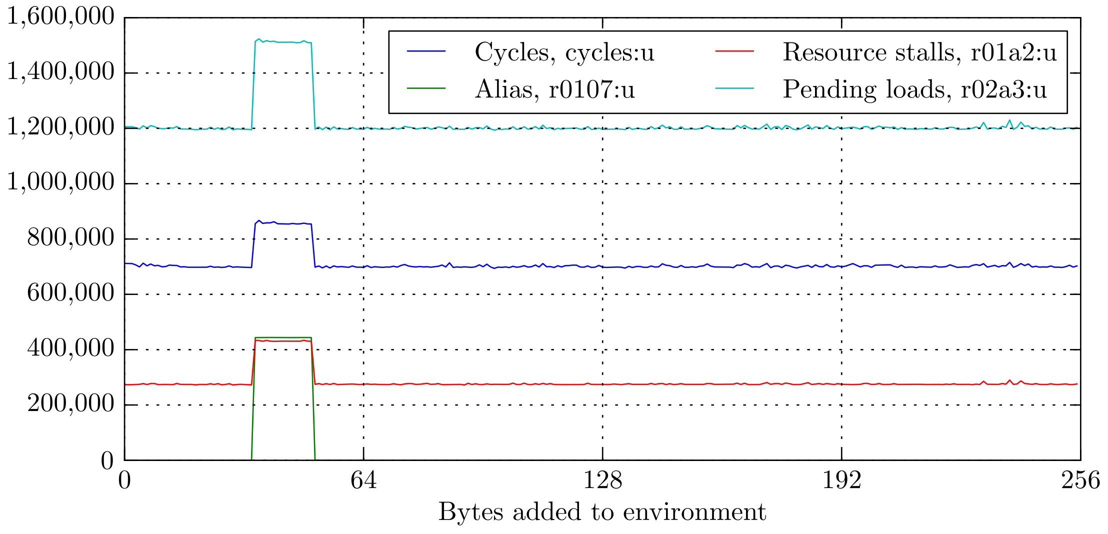

Figure 4.2: Cycle count correlating with resource stalls, pending memory loads and address alias performance counter events.

To find out which memory accesses collide, we need to know the memory addresses of each variable at run time. Referring to Listing 4.1; statically allocated variables `i`, `j` and `k` have their virtual addresses determined at compile time, and are not affected by offsetting stack address. Inspecting the symbol table section of the ELF object file[1], reveals that the addresses of i, j and k are 0x601028, 0x60102c and 0x601030, respectively.

```
Symbol table '.symtab' contains 66 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name

    42: 0000000000601028     4 OBJECT  LOCAL  DEFAULT   25 i
    43: 000000000060102c     4 OBJECT  LOCAL  DEFAULT   25 j
    44: 0000000000601030     4 OBJECT  LOCAL  DEFAULT   25 k
```

Variables `g` and `inc` are stack allocated local automatic variables, which will be affected by changing initial stack position. It is difficult to observe their correct addresses without introducing observer effects. Simply adding a call to printf for instance, completely changes the program behavior with respect to bias. Instead, we use a small amount of assembly code to calculate the addresses and print the numbers directly using system calls. We find that the spike in cycle count occurs when the address of `g` and `inc` are 0x7fffffffe028 and 0x7fffffffe02c, respectively.

[1]Output is generated using the readelf utility, part of GNU Binary Utilities

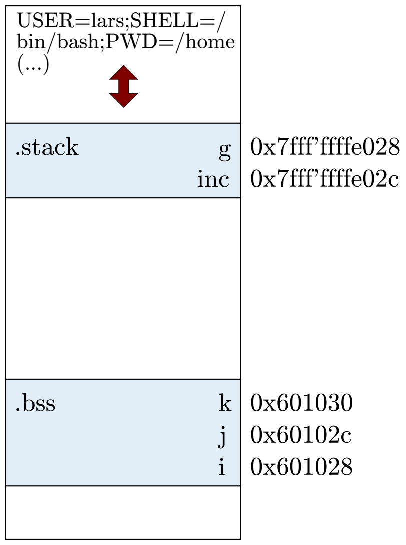

Figure 4.3: Stack allocated variables end up with the same address suffixes as static variables as a result of environment size.

Notice that the last twelve bits (three hexadecimal digits) for `g` and `inc` are the same as for `i` and `j`. The two aliasing pairs are illustrated in Figure 4.3.

$$\begin{aligned}(\mathrm{g}, \mathrm{i}) &\rightarrow (\mathrm{0x7fffffffe028}, \mathrm{0x601028})\\ (\mathrm{inc}, \mathrm{j}) &\rightarrow (\mathrm{0x7fffffffe02c}, \mathrm{0x60102c})\end{aligned}$$

There is a period of 16 byte of environment size where cycle count spikes (see Figure 4.2). This can be explained by a default stack alignment of 16 byte in GCC, meaning addresses of stack variables will only change in multiples of 16. It is therefore impossible to have only one aliasing pair of variables in this case. The next address suffixes of `g` and `inc` will be 0x0**3**8 and 0x0**3**c, aliasing with neither `i`, `j` or `k`. The effect is *periodic*; with increasing environment size, the spike in cycle count happens every 4096 byte.

Relation to Previous Architectures

When presenting similar results for the Core 2, the authors mentions the performance counter LOAD__BLOCK.OVERLAP__STORE, which is not available on Ivy Bridge, as a potential indication of what causes of bias effects [16]. The documentation states that this counter triggers on loads blocked by a preceding store due to a variety of reasons, including the following case:

> "*The load and store have the same offset relative to the beginning of different 4-KByte pages. This case is also called 4-KByte aliasing.*" [12]

It appears that effects of 4K aliasing emerge as a case of failed store forwarding in earlier architectures. Address aliasing between static and automatic variables was probably the underlying cause of bias in the original paper as well.

Avoiding Aliasing – A Proof of Concept

Addresses of automatic variables can not be determined statically, because the position of stack at runtime is generally unknown. In addition to being offset by environment variables, the stack address can also be perturbed by other factors such as address layout randomization. Although we can not easily know *if* a collision is going to happen for a given environment, we can try to change the program to account for possible alias effects. The following strategy is a proof of concept of how alias-free code can be generated in cases like this.

1. Variable addresses can be accessed in C, thus we can read out the addresses to check if there are any collisions between (g, i) and (inc, j).

2. If the addresses do alias, branch to an alternative but semantically equivalent code path. Allocate a new set of variables to avoid aliasing.

A complete implementation is shown in Listing 4.2. Note that only one pair of variables needs to be checked in this case. If inc and j alias (which is checked), then g and i will alias as well.

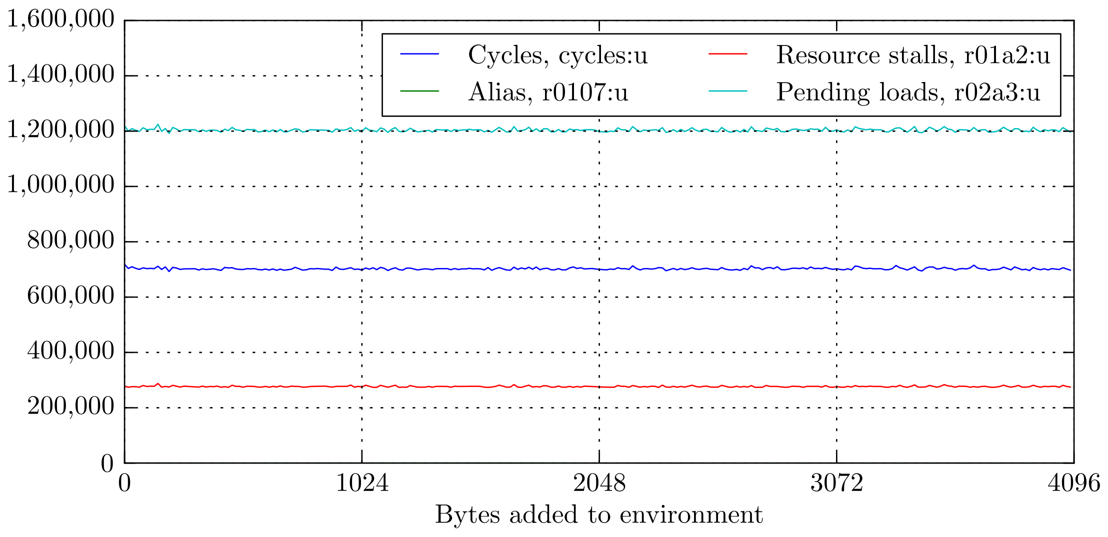

Figure 4.4: Performance counter statistics for modified code, showing stable values over a full 4 KiB range of possible stack addresses modulo 4096

Figure 4.4 shows that all performance counter statistics are stable through 4096 byte of offset. This covers a whole period of 12 bit address suffixes, showing that the

Listing 4.2 Modified code with explicit check for aliasing between inc and j. The alternative loop contains no aliasing, and this program does not suffer from bad performance due to address aliasing for any environment size.

```
static int i, j, k;

int main() {
    int g = 0, inc = 1;

    if ((((long)&inc) & 0xfff) == (((long)&j) & 0xfff))
    {
        int dummy = 0, t1 = g, t2 = inc;
        for (; t1 < 65536; t1++) {
            i += t2;
            j += t2;
            k += t2;
        }
        g = t1;
    }
    else
    {
        for (; g < 65536; g++) {
            i += inc;
            j += inc;
            k += inc;
        }
    }

    return 0;
}
```

aliasing is not merely moved to another offset. With only negligible overhead from additional code size and instructions executed, we are able to completely remove negative bias effects caused by stack position for this program.

4.1.2 Heap Address Aliasing

Address aliasing can be caused by conflicting pairs of load/store operations to any part of memory. The previous section looked specifically at an example of collision between stack variables and static data. Another important scenario to consider is collisions in dynamically allocated memory. In this section, we will look at how aliasing in heap allocated memory can impact performance. In particular, functions that operate on pairs of contiguous arrays can be vulnerable to 4K aliasing [10].

Many algorithms work in a “sliding window” fashion; reading some data from an input buffer, do some computation, and write the result to an output buffer. Sample usages that fit this pattern include:

- Copy or moving data
- Vector operations
- Image filtering

A typical worst case for this group of functions is when each buffer have the same addresses modulo 4096. For example, iterating over some data from a buffer starting at 0x601**020** and writing the result to another buffer starting at 0x865**020** is likely to generate aliasing. We will study one such program in detail, and show how variations in heap addresses can give performance variations of more than 50 %, even for compiler optimized programs. To provide a little more background, we will first look more closely at where dynamically allocated memory is located in virtual address space.

Addresses of Heap Allocated Memory

Whether or not heap areas alias will depend on properties of the memory allocator used. On our system, the default memory allocator is found in GNU libc [2]. Acquiring dynamic memory at run time is usually done by calling `malloc`, which takes a number of bytes to allocate and returns a pointer to that area.

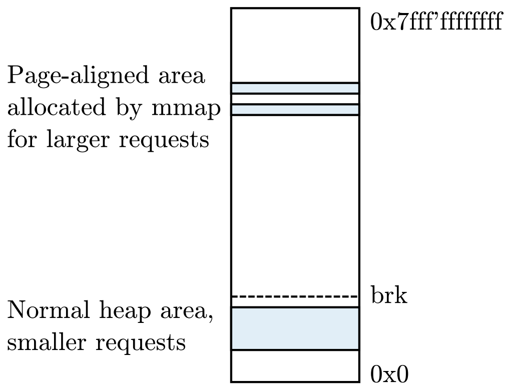
0x7fff'ffffffff
Page-aligned area allocated by mmap for larger requests
brk
Normal heap area, smaller requests
0x0

Figure 4.5: Heap memory spread out in virtual memory between `mmap` and `sbrk` requests

Depending on the *size* of the request, malloc uses two different strategies for how to allocate memory. For smaller allocations, malloc uses the “normal” heap area. As illustrated in Figure 2.3, the heap is placed after static code and data segments

[2]The GNU C Library (glibc) http://www.gnu.org/software/libc/

Table 4.2: Addresses of three consecutive heap allocations, for small and large request sizes.

Request size (B)	Heap addresses		
10	0x15410**1**0	0x15410**3**0	0x15410**5**0
1,000,000	0x7ff1b13ce**010**	0x7ff1b0e12**010**	0x7ff1b0d1d**010**

in virtual memory, extending upwards. Every process has a "break" position, which marks the heap address limit currently available to the process. The `sbrk` system call is used to increase the available heap space. For larger requests, heap allocation is typically done with `mmap`. This system call creates a mapping between memory address space and the contents of a file. Memory allocators use mmap with the `MAP_ANONYMOUS` flag specified for larger requests, creating an allocation that is not backed by a file. There is a threshold value `M_MMAP_THRESHOLD` for the minimum request size to allocate outside the normal heap, but mmap is also sometimes used for smaller requests than this value [3]. Memory returned by mmap is disjoint from the normal heap area, typically located on higher addresses and closer to the stack. Organization of different heap areas in virtual memory is illustrated in Figure 4.5.

One of the properties of mmap is that it guarantees page-alignment. Because the page size is 4096 bytes, memory returned from larger requests will *always* have alias on the last 12 bits – at least in all cases where mmap is used internally. Note that this property of malloc is not affected by address randomization. Table 4.2 illustrates the difference between addresses pointing to the normal heap or to areas allocated by mmap.

Example of Aliasing from Aligned Heap Areas

As an example of a "sliding window" program working on pairs of memory buffers, consider the function shown in Listing 4.3. It computes the convolution between an input array and a fixed kernel, writing the result to another array (endpoints skipped for simplicity).

This function is sensitive to aliasing between heap areas. Using malloc to allocate float arrays of size N, both input and output will alias in cases where mmap is used internally. For sufficiently large values of N, this will always be the case. Compiling with optimization O3 and N = 0x100000, a very large number of alias events and resource stalls are generated compared to cycle count. The performance counter statistics from running perf is shown in Table 4.3, under column "No padding".

Because malloc is using mmap and returns page-aligned data, a read from `input[i]` will always alias with a write to `output[i]` in convolve. Although distinct memory locations, the processor assumes there are dependencies between them.

[3]http://www.gnu.org/software/libc/manual/html_node/Malloc-Tunable-Parameters.html

Listing 4.3 Convolution kernel which is vulnerable to aliasing between input and output arrays.

```
static float kernel[5] = {0.1, 0.25, 0.3, 0.25, 0.1};

void convolve(int size, float *input, float *output)
{
    int i, j;
    for (i = 2; i < size - 2; ++i)
    {
        output[i] = 0;
        for (j = 0; j < 5; ++j)
            output[i] += input[i-2+j] * kernel[j];
    }
}

int main() {
    float *input  = malloc(N*sizeof(float));
    float *output = malloc(N*sizeof(float));
    convolve(N, input, output);
    return 0;
}
```

Table 4.3: Performance counter statistics for convolution kernel with two different heap address alignments. No padding is default page aligned buffers returned by malloc.

	No padding	Padding 16
Cycle count, cycles:u	12,024,235	5,570,808
Address alias, r0107:u	**9,195,816**	**1,175**
Resource stalls, r01a2:u	7,742,251	1,290,946
Instructions, instructions:u	11,166,765	11,166,802

Padding Data Aliasing can be accounted for in software by offsetting one of the arrays before calling convolve. The main function is modified slightly, to make sure address suffixes of `input` and `output` differ by a variable amount. We allocate some extra padding for output, and use pointer arithmetic to adjust the initial address.

```
int main() {
    float *input  = malloc(N*sizeof(float));
    float *output = malloc((N + x)*sizeof(float));
    convolve(N, input, (output + x));
    return 0;
}
```

The address difference modulo 4096 between input and output arrays now becomes $4x$ bytes. Plotting this for increasing values of x shows that virtually all aliasing is removed after offsetting with 12 or more elements. Figure 4.6 shows the amount of aliasing is steadily decreasing on larger offsets. The best parameter is found to be at a 16 or more elements difference (at least 64 bytes). Collecting the same performance counters with offset 16, we see that aliasing is almost eliminated, and number of resource stalls is significantly reduced. The results are shown in Table 4.3. By simply altering the default addresses given by malloc, we are able to reduce the cycle count by more than 50 %.

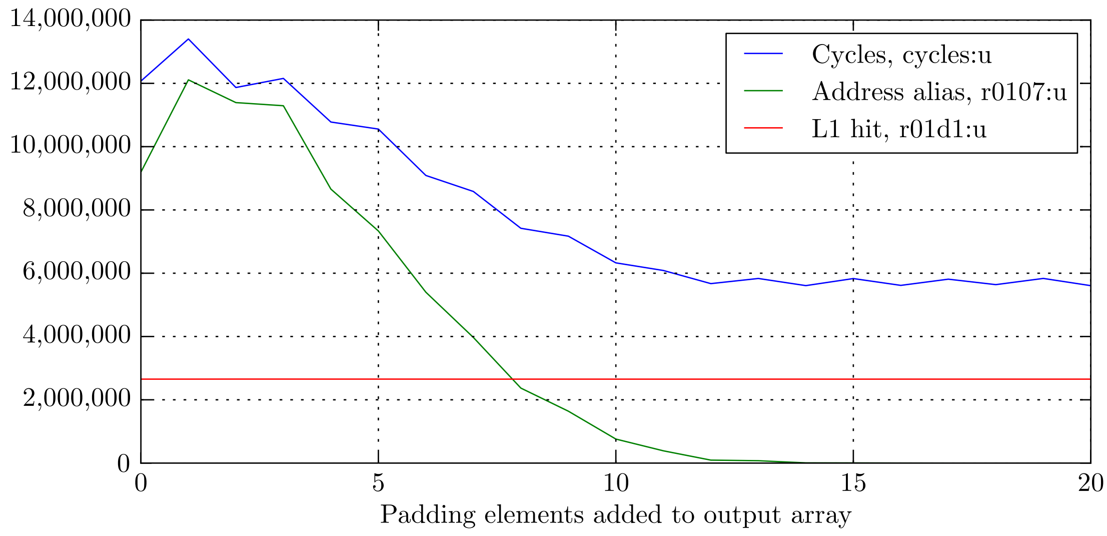

Figure 4.6: Performance statistics for convolution with variable amounts of padding for output array. `convolve(N, input, (output + x))` is executed for increasing values of x, separating array addresses by x `float` elements (or $4x$ bytes).

Cache Efficiency Given that we are working with relatively large amounts of data (more than what fits in L1 or L2 cache), one might suspect that cache misses impacts the cycle count. Measuring all the relevant cache related counters, as described in Chapter 3, we find that cache behavior appears to be completely unaffected by aliasing. L1 hits is shown in Figure 4.6, with no correlation to cycle count. We also measure hits in the Line fill buffer (LFB), L2 hits, LLC hits and LLC misses, but they all have stable values. Most of the memory access operations hit the L1 cache, which suggests the hardware prefetcher is able to keep up. No cache related events show any significant correlation to cycle count. We can therefore safely rule out cache as a contributing factor.

Architectural Optimization Address aliasing is an artifact of intricacies in the processor architecture. Ideally, one would like the compiler to create efficient code without having to manually offset heap pointers. On optimization O3 however, the compiler will optimize for a "generic" machine. One could hope that aliasing effects are considered when the compiler is asked to target our Intel architecture specifically, which can be done by specifying `-march=native`[4]. This includes using instructions that are not necessarily portable, for example AVX instructions.

We can provide additional information to the compiler's optimizer by using the `restrict` qualifier, explicitly stating that no pointer aliases exists. By marking input and output as restrict, we guarantee they are the only pointers to their respective memory areas, and not accessed through any other (aliased) pointers. This benefits code generation and optimization, potentially improving performance. Additionally, the input array should be specified as const, guaranteeing that only read operations are done to that array.

```
void convolve(int size, const float * restrict input,
   float * restrict output)
```

Together with the function signature changes, we find that the following compile parameters gives the best performance.

```
gcc -O3 -std=c99 -march=native
```

Compared to only using O3, the speedup is quite significant (as before, we use N = 0x100000). Specifying restrict seems to have the most impact on performance. Nevertheless, we still see a fairly high alias event count. As shown in Table 4.4, there are more than 250,000 alias events with the default heap alignment. Compared to the much lower cycle count, this is still a significant amount. Attempting to manually offset the output array again, we find the best parameter to be 48 or more (a minimum difference of 192 bytes). Separating the heap buffers eliminates almost all aliasing, while also significantly improving cycle count.

Even with every opportunity given to the compiler to account for aliasing, we are still able to squeeze out a speedup of 14 % by manually adjusting memory addresses. This speedup is also consistent through other input sizes.

[4]http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/i386-and-x86_002d64-Options.html

Table 4.4: Performance counter statistics for convolution example compiled with GCC, using modified function signature and optimal compiler optimization flags. Showing statistics for default (page aligned) heap addresses, and programmatically offset of output buffer by 48 `float` elements.

	No padding	Padding 48
Cycle count, cycles:u	3,451,031	3,028,162
Address alias, r0107:u	**259,605**	**236**
Resource stalls, r01a2:u	948,047	525,796
Instructions, instructions:u	3,302,429	3,302,444

Other Compilers In addition to GCC, we made similar experiments with LLVM's clang[5] version 3.0, and Intel's icc[6] version 13. The behavior of Clang is similar to GCC with respect to offsets to minimize alias. Only 16 padding elements is needed to get rid of aliasing, but the change in cycle count is not as clear. With almost ten times as many dynamic instructions executed, a much higher cycle count probably masks the added cost of aliasing. Code produced from Intel's icc is not free from aliasing on default page aligned heap memory either. A similar alias event count as GCC, about 260,000 for N = 0x100000, is generated with no padding. Aliasing can be eliminated almost completely, but that requires an offset of 160.

For this particular example, we found that GCC generated the most efficient code, executing in fewer cycles than both clang and icc. The effect of aliasing was also most significant in GCC, providing the most speedup when padding output buffer. Even when provided with architecture-specific optimization flags, none of the compilers we tested generated code that could not be improved by manually offsetting heap area.

4.1.3 Bias from Dynamic Libraries

Heap address conflicts discussed in the previous section does not show *measurement bias* per se, but rather an artifact of the particular implementation of malloc used. A perfectly legal implementation of malloc could insert offsets on every other large request, avoiding bad performance in cases when mmap is used internally. Variations between different versions of libc, or local configurations on parameters such as `M_MMAP_THRESHOLD`, could have a huge impact on performance. In general, one must consider dynamic libraries as an important part of the execution context. In this section, we will briefly study how some alternatives to the default allocator in libc behaves.

[5]clang: a C language family frontend for LLVM, http://clang.llvm.org/

[6]http://software.intel.com/en-us/intel-compilers/

Table 4.5: Addresses returned when allocating two `float` arrays of the same size, examples of possible sources of 4K alias conflicts highlighted. Tests were run with ASLR disabled for reproducible results, but we get similar results with randomization enabled.

Array length	Default (ptmalloc)	TCMalloc	Hoard
3840	0x603010 0x606c20	0x695**000** 0x699**000**	0x2aaaac000070 0x2aaaac004000
4660	0x603010 0x6078f0	0x69d**000** 0x6a2**000**	0x2aaaac010070 0x2aaaac014d38
65535	0x2aaaaaad1**010** 0x2aaaaab12**010**	0x6a7**000** 0x6e7**000**	0x2aaaac020**070** 0x2aaaac070**070**
16702650	0x2aaaab091**010** 0x2aaaaf049**010**	0x8b57**000** 0xcb2f**000**	0x2aaaac030**070** 0x2aaaafff0**070**

Alternative Memory Allocators

In the case of memory allocators alone, there are several alternatives to libc. One of the key issues different allocators tries to address is contention in a multi-threaded environment. Heap management is a natural bottleneck for multi-threaded applications, as all threads share the same address space. A naive allocator will force serialization of calls to malloc and free from different threads [3].

The implementation used in GNU libc is called ptmalloc, which is based on Doug Lea's malloc[7]. We will look at two alternatives, Hoard[8] and TCMalloc[9], both explicitly targeting multi-threaded code. Implementations of malloc and related functions are typically provided as shared libraries. Both Hoard and TCMalloc can be used as drop-in replacements for already compiled code using the `LD_PRELOAD` environment variable. Directories specified this way will be searched first by the dynamic linker, resolving references to malloc to the alternative library.

```
$ LD_PRELOAD=~/Allocators/Hoard/libhoard.so ./test
```

For each of the allocator implementations, we observed the addresses returned when sequentially allocating two equally large `float` arrays. Table 4.5 shows a comparison for different request sizes.

- The default implementation (ptmalloc) is consistently using the regular heap (non-aligned low addresses) for smaller allocations, and mmap for larger requests.

[7]Doug Lea, A Memory Allocator, http://g.oswego.edu/dl/html/malloc.html
[8]The Hoard Memory Allocator, http://www.hoard.org/
[9]Thread-Caching Malloc, http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html

- TCMalloc seems to be using the normal heap area exclusively. Interestingly, memory is aligned on page boundaries also for small sizes, always returning addresses ending in 0x000.
- Hoard also tends to align on page boundaries for larger requests, with addresses ending in 0x070. Unlike the other two, it looks like Hoard never utilizes normal heap area.

Each library produces very different results, and potential for address alias conflicts depends on the particular implementation of malloc used. There are several examples of cases where switching library can eliminate alias by changing address suffixes. For the convolution example, using ptmalloc or Hoard is very likely to outperform TCMalloc for arrays of length 4660.

Most allocators seem focus mostly on efficiency in a multi-threaded environment, and we find that none of the alternatives tested attempts to solve the aliasing problem for aligned arrays.

4.1.4 Summary

In this section we have studied how *address aliasing* can affect program performance under different memory layouts. This effect is caused by the way speculative and out-of-order memory operations are handled by the CPU, only considering the last 12 address bits to resolve conflicts load and store operations. We have shown how aliasing can cause measurement bias for the following cases:

- *Variations in environment size*: Analyzing the example provided in [17], we determined that collisions between stack variables and static data caused aliasing for certain stack positions. Variations in environment size changes virtual addresses of stack allocated data, and measurement bias observed by this can in some cases be attributed to address aliasing.
- *Properties of heap allocators*: Different versions or implementations of memory allocators can introduce measurement bias. We presented a code example with extreme sensitivity of data alignment, with more than 50 % performance variation between different memory layouts.

Any change to virtual memory layout of data can potentially introduce bias effects from address aliasing. We find that compilers, even when provided with architecture specific optimization flags, still generates code that suffers from aliasing. Typical heap allocators are found to often use page-aligned memory, which is the worst case with respect to aliasing for many algorithms. Despite recommendations from Intel, we are not aware of any allocators that specifically addresses this issue. We show how manual padding of variables, and alternative alias-free code paths, can be used to avoid aliasing at run time. This means that an understanding of 4K address aliasing is sometimes needed to achieve optimal performance.

4.2 The Loop Stream Detector

Typical software spends most of the time executing the same instructions repeatedly in a loop, where the same branches are taken, and the same instructions are fetched and decoded. The Loop Stream Detector (LSD) is a front-end hardware optimization that is able to detect small software loops [20]. Instead of repeatedly fetching and decoding the same instructions, loops can be streamed directly from a queue of already decoded micro operations. Normal fetch/decode operation is resumed once a branch mispredict occurs.

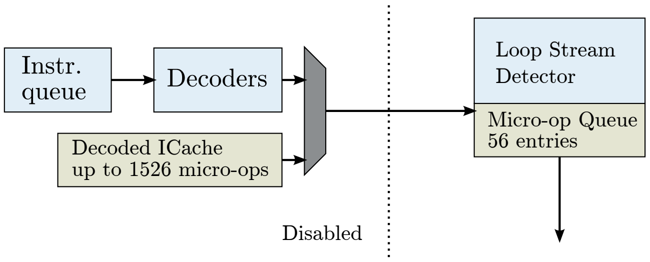
Instr. queue
Decoders
Decoded ICache up to 1526 micro-ops
Disabled
Loop Stream Detector
Micro-op Queue 56 entries

Figure 4.7: Loop Stream Detector circumvents branch prediction, fetch and decode for hot loops, streaming directly from the micro-op queue

This optimization is affecting both performance and power efficiency. Power savings come from disabling front-end components such as fetch and decode when the LSD is active. In cases where instruction fetch is the bottleneck, streaming already decoded micro-operations can also increase performance.

In the documentation, we find two hardware performance counters that can measure the utilization of the LSD [12].

LSD.UOPS Counts the number of micro-ops delivered by loop stream detector.

LSD_OVERFLOW Counts number of loops that can't stream from the instruction queue.

Note that none of these are listed in the non-architectural counters supported by Ivy Bridge. However, we find that using the event codes given for previous architectures seems to work regardless, and gives reasonable results. We suspect that the official reference is either not complete or misleading in this case.

We choose to study this hardware feature specifically, because it has been suggested in previous work as a potential source of measurement bias [17]. The Loop Stream Detector is dependent on code layout, which can change with external factors such as variations in link ordering. Interacting with the alignment and position of instructions in memory, the LSD can cause performance differences for programs with otherwise *identical* instruction sequences.

Some effort has been made to model properties of the LSD on previous Intel architectures [8], as an aid in low-level optimization. In this section we will uncover important characteristics of the Loop Stream Detector on Ivy Bridge. With an understanding of its limitations, we show an example of how measurement bias from link ordering can be explained. We also discuss other external factors that can alter code layout, possibly triggering performance variations from interacting with the LSD.

4.2.1 Properties of the Loop Stream Detector

Loops can only be recognized by the Loop Stream Detector under certain conditions. The Optimization Manual lists the following restrictions for the "Sandy Bridge" Loop Stream Detector [10]:

1. Up to eight chunk fetches of 32 instruction-bytes
2. Up to 28 micro-ops (~28 instructions)
3. All micro-ops are also resident in the Decoded ICache
4. Can contain no more than eight taken branches and none of them can be a CALL or RET
5. Cannot have mismatched stack operations. For example, more PUSH than POP instructions

The capabilities of the LSD is changing between processor generations. In particular, the limits to chunk fetches, maximum number of micro-ops and taken branches seem to continuously increase [20, 1]. An increase in the limit of micro-ops, from 28 to 56, is listed as an enhancement to the Ivy Bridge front-end over Sandy Bridge [10]. It is *not clear* whether the remaining limitations from Sandy Bridge are valid for the Ivy Bridge, as it is not explicitly stated in the manual.

Table 4.6: Properties of the Ivy Bridge Loop Stream Detector, determined empirically by a series of micro benchmarks.

	HT off	HT on
Chunk fetches	12	12
Micro-ops	56	28
Taken branches	12	12

We create a series of micro benchmarks to manually verify what the actual limitations are. The updated Loop Stream Detector limitations for Ivy Bridge is summarized in Table 4.6. We find that the limitation to chunk fetches, as well as taken branches, has increased from previous generations.

Chunk fetches

The documentation states that a maximum number of eight 32 byte “chunks” of instruction data can be fetched in each loop. Our interpretation of this is that instruction data is somehow organized into blocks of 32 bytes, aligned to multiples of 0x20 in virtual address space. To test this limit, we need loops that execute instructions spanning a variable amount of chunks. The micro kernel we used is shown in Listing 4.4, written in GNU x86-64 assembly.

Listing 4.4 Micro kernel for testing chunk fetch limit. Additional (`cmp`, `je`, `lea`, `lea`, `lea`, `add`) blocks of exactly 32 bytes are inserted until LSD.UOPS shows the Loop Stream Detector is no longer active.

```
    .section .text
    .globl main
main:
    mov $0x1234567, %eax
    mov $0, %ebx
    .p2align 5
.l0:
    cmp $0, %ebx
    je .l1
    lea 0x100(%rsp), %ecx
    lea 0x100(%rsp), %ecx
    lea 0x100(%rsp), %ecx
    add $0x100, %ecx
.l1:
    (...)
.l11:
    cmp $0, %eax
    jne .l0
    ret
```

The idea is to execute a minimal amount of instructions at the start of each block, and immediately jump on to the next. Blocks are labeled `.l0`, `.l1`, . . . , `.ln`, and the first block is explicitly forced to align on a 32 byte boundary (5 bits) by the `.p2align` directive. Virtual addresses of code and static data can be read from the ELF binary after compilation. A disassembled portion of the code is shown below[10].

```
400500: 83 fb 00             cmp    $0x0,%ebx
400503: 74 1b                je     400520 <.l1>
400505: 8d 8c 24 00 01 00 00 lea    0x100(%rsp),%ecx
40050c: 8d 8c 24 00 01 00 00 lea    0x100(%rsp),%ecx
```

[10]Output is generated by the objdump utility, part of GNU Binutils

```
400513: 8d 8c 24 00 01 00 00 lea    0x100(%rsp),%ecx
40051a: 81 c1 00 01 00 00    add    $0x100,%ecx
```

We see that the first compare instruction starts on address 0x400500, aligned to 32 byte. Because `%ebx` is always zero, the branch is taken every time. The remaining `lea` and `add` instructions are added to fill the remaining bytes in the chunk. Only the first $3 + 2$ instruction bytes occupied by `cmp` and `je` are executed, while the remaining 27 bytes is dead code. After a number of repeated blocks like these, the counter is incremented before jumping back to `.l0`. The final `sub`, `cmp` and `jne` instructions occupy 12 bytes, so it is possible to insert another 20 bytes of padding to completely fill the last chunk as well.

```
400674: 83 e8 01             sub    $0x1,%eax
400677: 83 f8 00             cmp    $0x0,%eax
40067a: 0f 85 80 fe ff ff    jne    400500 <.l0>
```

For every loop iteration, a total of $2N + 3$ instructions are executed, spanning $N + 1$ chunks of 32 bytes. The `cmp` and `je` instructions should be fused to a single micro-op, so we should not be hitting the micro-op limit. We find that the LSD.UOPS counter reports high values through $N = 11$, meaning the Ivy Bridge Loop Stream Detector must support at least 12 chunk fetches.

Taken branches

The fourth requirement states that a loop can contain no more than eight taken branches. The chunk fetch limit example already uses 12 branches, so we can immediately conclude that at least 12 taken branches is supported. We experiment with more than one branch target in each chunk, but are not able to increase the number any further. All of our example programs with more than 12 taken branches results in low values for LSD.UOPS.

Micro-operations

The second requirement states that each loop can only contain a maximum of 28 micro-operations. This is no longer true for Ivy Bridge, because of changes to the Micro-op Queue. In previous generation(s), the micro-op queue was statically partitioned with 28 entries for each logical core. One of the Front-end enhancements in Ivy Bridge is that all 56 entries in the queue can be used when only a single logical core is active, i.e. hyper-threading is disabled [10].

We verify this restriction by measuring LSD_OVERFLOW and LSD.UOPS for loops of increasing size, adding instructions one by one until the performance counter statistics report the LSD is inactive. Our micro-kernel is shown in Listing 4.5. We find that the LSD is active with as many as 54 `add` instructions. The loop logic is implemented with a `sub`, `cmp` and `jne` instruction. With macro-fusion of the last

Listing 4.5 Micro kernel for testing LSD micro-instruction limit. Additional `add` instructions are inserted until LSD.UOPS shows the Loop Stream Detector becomes inactive.

```
    .section .text
    .globl main
main:
    mov $0x1234567, %eax
    .p2align 5
.loop:
    add %ebx, %ecx
    add %ebx, %ecx
    (...)
    add %ebx, %ecx
    sub $1, %eax
    cmp $0, %eax
    jne .loop
    ret
```

`cmp` + `jne`, the total is exactly 56 micro-operations per loop. With hyper-threading enabled, the number is cut in half to 28, as expected.

4.2.2 Hitting the Chunk Fetch Limit

Knowing the parameters, we can construct programs that are on the limit of what the Loop Stream Detector accepts. Listing 4.6 shows a simple loop that generates code much like the micro kernel used to test chunk fetch limit. Each of the if statements occupies exactly 32 byte in instruction memory when compiled with gcc and no optimization. Including the instructions needed to compare and increment variable `i`, the loop code covers just under 12 × 32 bytes in the compiled binary.

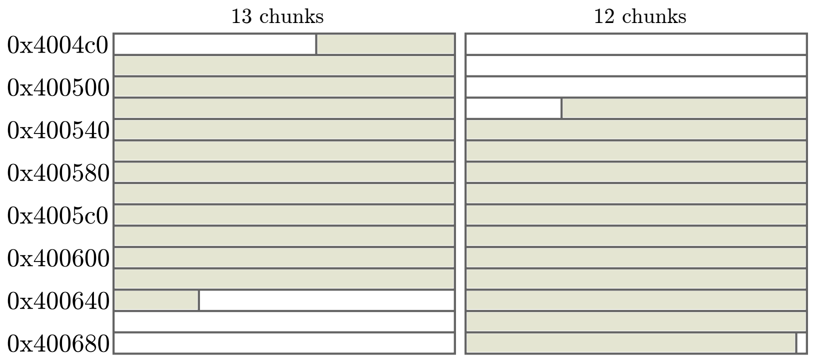

Figure 4.8: Depending on alignment, code can span either 12 or 13 chunks of 32 byte instruction memory

Listing 4.6 Loop spanning 13 chunks of 32 bytes when compiled with gcc, but only 12 chunks with the printf statement uncommented.

```
#include <stdio.h>
#define B27 a=b, a=b, a=b, a=b, a=b, a=b, a=b, a=b, a=b
int main() {
    register int i = 0, a = 1, b = 1;
    //printf("What is going wrong?");
    while (i++ < 0x12345678) {
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
        if (a != b) B27;
    }
    return 0;
}
```

Even though we are within the limit, a quick test with `perf` reveals that the Loop Stream Detector is not enabled. Only a negligible amount of micro-ops are delivered by the LSD, as reported by LSD.UOPS. Inspecting the compiled binary, we see that the loop instructions span virtual addresses 0x4004d3 through 0x400648. The first byte in a chunk have address ending in 0x00, 0x20, 0x40, 0x60, 0x80, 0xa0, 0xc0 or 0xe0, meaning the first instruction belongs to the 0x4004**c0** chunk, and the last is just within the 0x4006**40** chunk. Despite covering less than 12 $\times$ 32 byte of instruction addresses, this specific alignment causes the code to span not 12 but 13 chunks, exceeding the limitation.

We can create a different alignment by inserting some code before the loop. By uncommenting the printf statement, we see that the LSD is able to deliver micro-ops, giving a huge speedup. The performance counter statistics for each of these programs is shown in Table 4.7.Looking at the binary again, we find that the loop now only covers 12 chunks, spanning addresses 0x400529 through 0x40069e. The difference between the two configurations is illustrated in Figure 4.8. When no optimization flags are used, GCC does not try to align loops in any strict way, meaning that differences like these depend on surrounding code. We will use a similar approach to show how bias from link ordering can be explained by the LSD, presenting an example that also works with optimization enabled.

Table 4.7: Performance counter statistics for loop spanning 12 or 13 chunks of instruction memory.

	13 chunks	12 chunks
Cycle count, cycles:u	7,636,642,008	3,665,689,143
LSD.UOPS, r01a8:u	**10,921**	**4,581,309,568**
Instructions, instruction:u	8,246,427,939	8,246,428,712

4.2.3 Bias from Link Order

Different link orders affects the relative position of code and static data within the final compiled binary file. Interacting with limitations of the Loop Stream Detector, variations in instruction addresses can *bias* programs towards certain link orders. We will look at an example of exceeding the chunk fetch limit depending on link order, which works on GCC with optimization O3. Bias from hitting limitations of the Decoded ICache is also discussed.

Chunk fetches

With a few modifications to our previous example, we can create a program that utilizes the LSD only for certain link orders. Our program is shown in Figure 4.7, containing three files; main.c, foo.c and loop.c.

In the example from Listing 4.6, we exploited weak alignment guarantees from GCC, sometimes spilling loop code over 13 chunks. One might hope that with the proper optimization flags, compilers will fix such alignment flaws. There is an "align-loops" flag in GCC[11], enabled by default in O2 and O3, which allows the assembler align loops by inserting necessary padding. On optimization O3, the following assembly directives are inserted before the first instruction that is part of the loop in loop.c[12].

```
.p2align 4,,10
.p2align 3
```

The first directive tries to align to four address bits, inserting at most 10 bytes for padding. If that does not work, the next directive forces loops to be aligned to three bits – meaning addresses ending in 0x0 or 0x8. Within a 32 byte instruction block, the first loop instruction can end up being offset by 0x00, 0x08, 0x10 or 0x18. The loop itself spans 379 instruction bytes when compiled in our example, just shy of the $12 \times 32 = 384$ bytes that can maximally fit in the LSD. Out of the four possible

[11]Optimize Options – Using the GNU Compiler Collection (GCC), http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[12]The GNU Assembler, http://tigcc.ticalc.org/doc/gnuasm.html

Listing 4.7 Program with measurement bias from link order, favoring (main, foo, loop) over (main, loop, foo) when compiled with GCC and optimization level O3

foo.c:

```
int foo(int n)
{
    int a = 1;
    return a;
}
```

main.c:

```
#include <stdio.h>

int main()
{
    int i = loop(10);
    int f = foo(10);

    printf("Loop: %d\n",i);
    printf("Foo : %d\n",f);

    return 0;
}
```

loop.c:

```
#define B22(n) b += i*i*a + 1
volatile int i = 42;

int loop(int a) {
    register int b = i==42 ? 1 : 0;
    do {
        if (!i) B22(1);
        if (!i) B22(2);
        if (!i) B22(3);
        if (!i) B22(4);
        if (!i) B22(5);
        if (!i) B22(6);
        if (!i) B22(7);
        if (!i) B22(8);
        if (!i) B22(9);
        if (!i) B22(10);
        if (!i) B22(11);
      } while (i++ < 0x12345678);
    return b;
}
```

alignments, only zero offset into a chunk will work in our case. Performance counter statistics for two different link orders are shown in Table 4.8.

- Link order *main.c, loop.c, foo.c*: The first loop instruction is aligned to 0x400590, an offset of 0x10 into the 0x4005**80** chunk. Covering 13 chunks, the Loop Stream Detector can not be utilized.

- Link order *main.c, foo.c, loop.c*: The first loop instruction is aligned to 0x4005**a0**, right at the start of a chunk. Cycle count is reduced by more than 50% from utilizing the LSD.

We see that measurement bias from link order can in some cases be explained by hitting the chunk fetch limit of the Loop Stream Detector. Even when compiling with sensible optimization flags, GCC does not always hit the optimal loop alignment, causing code to spill over additional chunks. A possible solution in this scenario could be to manually specify the loop alignment to 32 byte, with -falign-loops=32.

Table 4.8: Performance counter statistics for different link orders.

	main.c, loop.c, foo.c	main.c, foo.c, loop.c
Cycle count, cycles:u	7,636,374,873	3,665,668,196
LSD.UOPS, r01a8:u	**10,998**	**8,246,350,119**
Instructions, instruction:u	11,606,047,809	11,606,046,865

Decoded ICache Limitations

Any property of the Loop Stream Detector that depends on the code layout in memory can potentially be triggered by different link orders. This is not limited to the number of chunk fetches, but also includes properties of the Decoded ICache. One of the LSD requirements states that all micro-ops must be present in the Decoded ICache. The manual describes additional restrictions to what micro-ops can be stored in this cache, imposing limitations to the number of branches per block, and other alignment properties [10]. Consider the C program from Listing 4.8.

Listing 4.8 Loop using LSD only for certain alignments when compiled with GCC and no optimization, likely hitting limitations in Decoded ICache

```
#include <stdio.h>
volatile static int color;

int main() {
    printf("Fixing colors");
    for (int i = 0; i < 10000000; ++i) {
        if (color == 0x000)
            color = 0xaaa;
        else if (color == 0xaaa)
            color = 0xfff;
        else if (color == 0xfff)
            color = 0x000;
    }
    return 0;
}
```

Compiling with `cc -std=c99` and no optimization, the binary code generated by this function spans far less than 12 chunks. Counting LSD.UOPS, we see that the Loop Stream Detector is not in use. However, by removing the printf statement the loop alignment is moved from 0x400517 to 0x4004c1, and the LSD is enabled. We did not investigate the actual cause of this any further, but speculate that specific alignments prevents all micro-ops to reside in ICache simultaneously.

4.2.4 Other Triggers of Bias Effects

The focus of previous work, and elaborated in this section, is measurement bias caused by altering link ordering. The important factor is not the process of linking itself, but the effect it has on final code layout in the compiled binary. Any interaction that affects code layout can potentially trigger performance cliffs caused by the LSD. The following briefly discusses some additional properties to consider.

The order of functions within a source file

As with link order, the order of functions within a source file usually determines their relative position in the text segment of the compiled binary. Given a program with functions foo and bar, listing foo before bar and vice versa generates two "different" programs with respect to memory layout. GCC prints functions to the text segment in the order they appear in the source code.

Length of external symbols

Symbols such as function names are placed in a symbol table section in the ELF file. When running readelf or objdump, the names are used to print nice output. Symbols that needs to be resolved dynamically are allocable, and mapped to virtual address space *before* the text segment. This means that names of external symbols resolved at run-time does affect code layout. Longer function names (or more external dependencies) will offset the text segment because the size of allocable segments related to dynamic linking changes. This includes the "dynsym", "dynstr", and "rela" sections in an executable made with GCC. Details for each of the ELF sections can be found in the man pages [13].

Some particularly devious "bugs" can occur from this. Consider a scenario where a program does not align correctly to utilize the Loop Stream Detector. For some debug purpose, assume a printf statement is added to a completely unrelated part of the code. The additional external symbol offsets instruction alignment such that the LSD can be used, giving a huge speedup. Later removing the printf statement will reduce the size of external dependencies, and result in worse performance again.

Interpreter path

When executing an ELF binary, the first thing that happens is to load and run an interpreter/dynamic linker. The interpreter is responsible for unpacking the object file and map all the allocable sections to virtual memory, as well as loading any external dependencies. The path to the interpreter itself is located in the ELF ".interp" section, which is allocable and loaded first. On our system, this defaults to "/lib64/ld-linux-x86-64.so.2". Other loaders can be specified, either at compile time

[13]http://linux.die.net/man/5/elf

or by patching the object file[14]. Specifying an alternative interpreter at compile time can be done with a linker flag, for example:

```
-Wl,-dynamic-linker,/home/me/very/long/path/to/an/alternative
   /loader.symlink.so.2
```

The string containing the path to the interpreter is actually allocated to virtual memory, before the text segment. This means that the string length (in characters) can offset code addresses. This bias trigger is more of a curiosity, and probably not a very likely scenario in practice.

4.2.5 Summary

The Loop Stream Detector can provide significant speedup, but subtle changes to code layout sometimes prevents this optimization to be utilized. In this section we have shown that simply adding (or removing) a printf statement, or changing the link order, can have a severe performance impact. Programmers rarely care, or even know, about intricate details such as the addresses and alignment of code. We expect, and probably rightfully so, that sensible compilers and linkers will organize code somewhat optimally. In cases where this is not true, weird effects that might be categorized as “bias” occurs. For developers, it is useful to be aware of the fact that *any* change to instruction addresses can potentially have a huge impact on performance.

[14]http://nixos.org/patchelf.html

Chapter 5

Case Studies

In this chapter, we show how bias effects discussed previously can affect real software. The goal is to identify possible instances of bias, and also investigate how to apply new knowledge of these effects for optimization purposes. We choose to focus on already highly optimized numerical applications for our case studies, because the real world impact of bias is most relevant for performance critical applications. If only a minor improvement can be made by avoiding some memory alignment issue, the cost of implementing an architecture specific optimization in a high performance numerical library will probably be worth it.

Two applications are studied: In Section 5.1 we will look at FFTW [7], a widely used library for computing discrete Fourier transforms. We show that for smaller input sizes, there is potential for bias towards certain stack positions. In Section 5.2 we look at ATLAS [28], which is an implementation of the BLAS API [4] for linear algebra routines. We find that matrix-vector multiplication is sensitive to address aliasing, and show how to significantly improve worst case performance by padding data.

5.1 FFTW

As an integral part of applications such as signal and image processing, a huge amount of work has been done over the years to optimize and tweak the performance of the Fast Fourier Transform (FFT). Today, there are a plethora of excellent implementations available, among them FFTW[1], an acronym for "Fastest Fourier Transform in the West". Its design goal is to be portable, yet achieve close to optimal performance across a wide variety of platforms [7]. The library is not optimized specifically for any processor or architecture, but uses *automatic tuning* to adapt to the underlying hardware [24]. Provided that a speedup on average can be achieved, explicitly handling of context bias could be a realistic addition to FFTW.

[1]FFTW Home Page,http://www.fftw.org/

Listing 5.1 C program snippet for computing the Fourier transform of N double precision complex numbers X times.

```c
int main()
{
    fftw_complex *in  = fftw_malloc(sizeof(fftw_complex)*N);
    fftw_complex *out = fftw_malloc(sizeof(fftw_complex)*N);

    fftw_plan p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD,
       FFTW_ESTIMATE);

    for (int i = 0; i < X; ++i)
        fftw_execute(p);

    fftw_destroy_plan(p);
    fftw_free(in), fftw_free(out);
    return 0;
}
```

Installation and Configuration We used the currently most recent version 3.3.3 of FFTW in all our tests, compiled from source and built as a shared library. The configure script is invoked with the following parameters:

```
./configure CC="gcc -march=native" --enable-shared --enable-
   sse2 --enable-avx
```

For optimal performance on our machine, we explicitly enable support for SSE2 and AVX instructions. In addition, we set the “arch” compiler flag to “native”, allowing the GCC to specifically tune generated code to our architecture.

Setup and Methodology FFTW is a very comprehensive library with lots of functionality. Our approach will be to concentrate on smaller transform sizes, and identify specific examples where bias is easily measurable. We create a small C program, shown in Listing 5.1, with two parameters; N, the number of elements to transform, and X, the number of times to repeat the transform. The iteration count is used to amplify any bias for smaller input sizes. The API calls made in the example are explained below.

fftw_malloc is a wrapper around the standard malloc, returning heap-allocated data. This is supposed to provide a stronger alignment guarantee than malloc does, needed for vectorized code. The system's default malloc might not align data on wide enough boundaries, although reasonable implementations do.

fftw_plan_dft_1d creates a *plan* for how to most efficiently compute a discrete Fourier transform of size N. The plan is a recipe for how to decompose the work into several kernels. A *kernel* is a function that can compute the Fourier transform for one particular input size. Kernel implementations are automatically generated C programs, which are also called *codelets*. The planning step encapsulates the auto-tuning part of FFTW, as plans can be constructed based on actual measurements and benchmarks done on the current machine. Knowledge of well performing plans can be stored as *wisdom*, which is used to construct other plans later.

fftw_execute performs the computations specified by the given plan, calculating the Fourier transform of "in" and writing the result to "out".

This setup is used to identify sensitivity to, or bias from, changes in *environment size*. The assumption is that stack alignment when calling fftw_execute can potentially introduce bias. This program resembles the example we studied in Chapter 4, where address aliasing caused bias for certain environment sizes. Specifically looking for a similar effect here, the most relevant performance metrics are cycle count and address aliasing (r0107:u). For all our experiments, we use the same machine and methodology as outlined in Chapter 3.

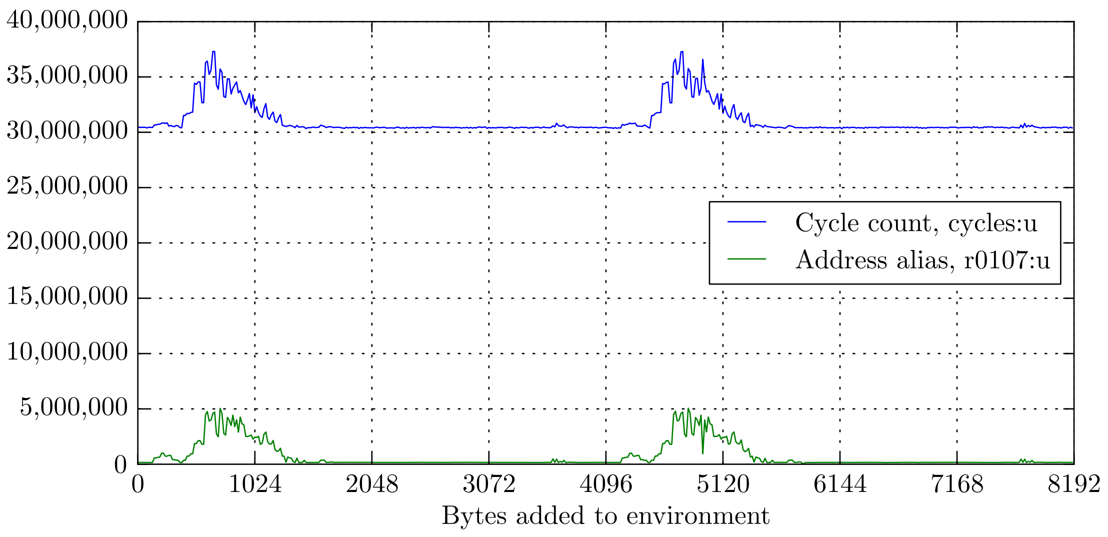

Figure 5.1: Performance counter measurements for computing complex DFT of size $N = 16$, repeated $X = 200{,}000$ times. Address aliasing under some environment sizes negatively impacts cycle count.

Environment Bias in Small Transforms A quick search through possible configurations is done manually, by specifying different values for iteration count and transform size. We find that computing transforms of size 16 have noticeable bias. Figure 5.1 shows the relation between address alias and cycle count for repeated benchmarks under increasing environment size. The effect is amplified by executing the same plan $X = 200{,}000$ times, resulting in significant performance variations.

We choose study this particular case in depth, with the purpose of finding out why aliasing occur and how it can be avoided. The call graph and generated assembly code is analyzed in detail in Section 5.1.1, to identify where aliasing occur. In sections 5.1.2 and 5.1.3 we discuss two possible optimizations that can be made to avoid aliasing. The first describes possible improvements to the implementation of FFTW itself, while the second option is to account for aliasing at the user level. Bias is not exclusively a problem for this particular input size, and we show how similar patterns also occur in other kernels.

5.1.1 Analysis of Aliasing Effects

The first step in determining the cause of aliasing is to look at exactly what code is executed. An overview of which kernels are used to execute a plan can be found by calling fftw_print_plan. In the case of input size $N = 16$, the plan consists of a single kernel called “n1fv_16”, compiled with AVX instructions enabled. Valgrind [2] is used to analyze the call graph from fftw_execute, shown in Figure 5.2. There is a chain of calls, eventually leading to the actual kernel invocation.

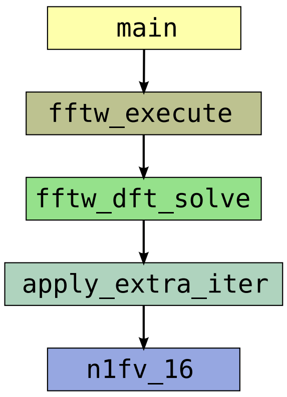

Figure 5.2: Call graph showing the actions taken after calling fftw_execute for the 16 element transform.

The kernel itself is located in the file dft/simd/common/n1fv_16.c, which contains an automatically generated C program for computing a 16 element DFT[3]. The function signature gives an idea of how the kernel works. There are 9 parameters, where the most interesting are input buffer (“ri”) and output buffer (“ro”).

[2]http://valgrind.org/

[3]There are actually two separate implementations provided, one specifically optimized for architectures supporting fused multiply-add instructions. FFTW is compiled with FMA disabled in our case, as there are no such instructions on Ivy Bridge.

```
void n1fv_16(const R *ri, const R *ii, R *ro, R *io, stride
   is, stride os, INT v, INT ivs, INT ovs)
```

Conflicting memory accesses within this function are causing bias effects. Looking at the generated assembly, we find three areas of memory whose address suffixes can potentially overlap:

- Heap allocated input and output buffers, parameters "ri" and "ro" respectively. These are pointers previously returned by calls to fftw_malloc, addresses in general unpredictable by the kernel, except for some alignment constraints.

- Stack allocated function parameters and local automatic variables. Compiled with optimization, the six first arguments will be passed in registers. The remaining three are pushed on stack.

- Statically allocated constants in memory. Three floating point constants are statically built into the object file, and their locations in memory are determined at runtime by the dynamic linker.

The static constants are loaded initially, using only a few load instructions. Because most memory access instructions are either to temporary stack variables or to heap, it is a fair assumption that address alias happens when stack accesses collide with heap addresses. Under synthetic testing with address randomization disabled, all parameters are fixed and deterministic. For our particular test, the heap allocated input and output buffers always reside in virtual addresses 0x602040 and 0x601c0 respectively – on the low end of a 4K segment of suffixes between 0x000 and 0xfff.

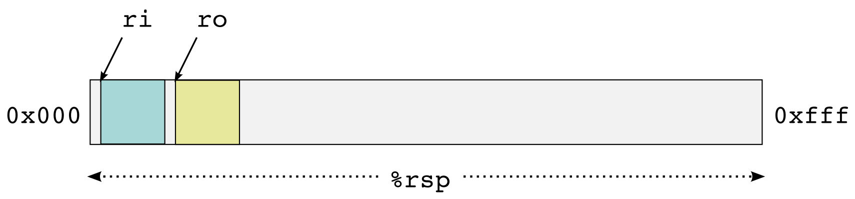

Figure 5.3: Position of stack (%rsp register) can be aligned anywhere within the space of virtual addresses modulo 0x1000. With ASLR disabled, the last 12 bits of `ri` and `ro` are 0x040 and 0x1c0 respectively.

As illustrated in Figure 5.3, the potential for overlap between stack and heap accesses will depend on where the stack is aligned, and the position of stack will vary with environment size. Alias effects likely occur when addresses of stack variables overlap with heap accesses. The program becomes *biased* to certain environment sizes, as it can execute with a "good" configuration with few collisions, or a "bad" configuration as in one of the periodic peaks illustrated in Figure 5.1.

5.1.2 Eliminating Bias at the Kernel Level

Assuming aliasing occurs from conflicts between stack and heap, bias can be eliminated by altering addresses of memory accesses to either of these areas. Once fftw_execute is called, the only thing we can control is the placement of stack allocated variables. Our solution idea is to programmatically adjust the stack in such a way that accesses do not collide (alias) with heap memory. We outline a naive solution, which exploits the fact that heap addresses are fixed with ASLR disabled. In our case, both heap allocated input and output buffers have address suffixes close to 0x000. A reasonable placement of stack is then on the other side of the spectrum, close to 0xfff and expanding towards lower addresses. The concept is illustrated in Figure 5.4.

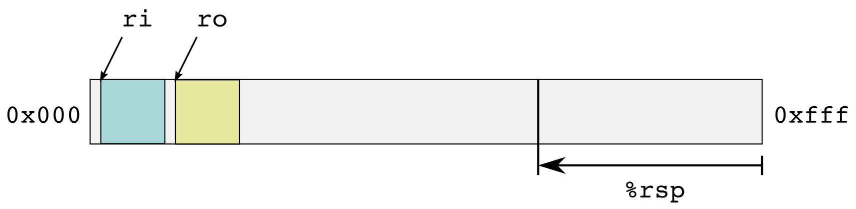

Figure 5.4: Collisions can be avoided by reseting the stack pointer to an address that does not conflict with the heap allocated areas.

Looking at the call graph from Figure 5.2 again, we have several options for where to programmatically align stack on a suitable address. In order to be transparent to the user, the adjustments must happen in either fftw_execute, fftw_dft_solve, apply_extra_iter or n1fv_16. Modifying fftw_execute or fftw_dft_solve will affect execution of *all* DFT plans, while we only want to target aliasing within a single kernel. A fix could be applied to n1fv_16 directly, but we choose to modify apply_extra_iter instead. As the name suggests, this function actually calls the kernel twice.

The modified function is shown in Listing 5.2. The `andq` instruction will zero out the last 12 bits of the stack pointer (%rsp register), effectively subtracting some number between 0 and 4096. The stack grows downwards, thus subtracting any (reasonably small) amount will not overwrite other data. We run the same benchmark again, with the results shown in Figure 5.5. We see that the modified version is much less sensitive to stack placement, removing almost all the bias from previously. Table 5.1 shows a reduction in cycle count by 11.8 % for the worst case, but unfortunately there is no significant speedup on average.

Even though we are able to remove most of the alias effects, performance did not improve as much as we had hoped. The constant overhead added by inserting the extra instructions in apply_extra_iter consumes any aliasing improvement. About 1 million dynamic instructions are added to our example in the modified version, an increase of 1.3 %.

Listing 5.2 Modified apply_extra_iter, aligning stack to a 4K boundary. The hidden function body makes two calls to the n1fv_16 kernel. Implementation is located in the file dft/direct.c.

```
static void apply_extra_iter(const plan *ego_, R *ri, R *ii,
   R *ro, R *io)
{
    void *rsp;
    asm volatile (
        "movq  %%rsp, %0;"
        "andq  $-4096, %%rsp;"
        : "=r"(rsp) : : );

    /* original implementation */

    asm volatile (
        "movq  %0, %%rsp;"
        : : "r"(rsp) : );
}
```

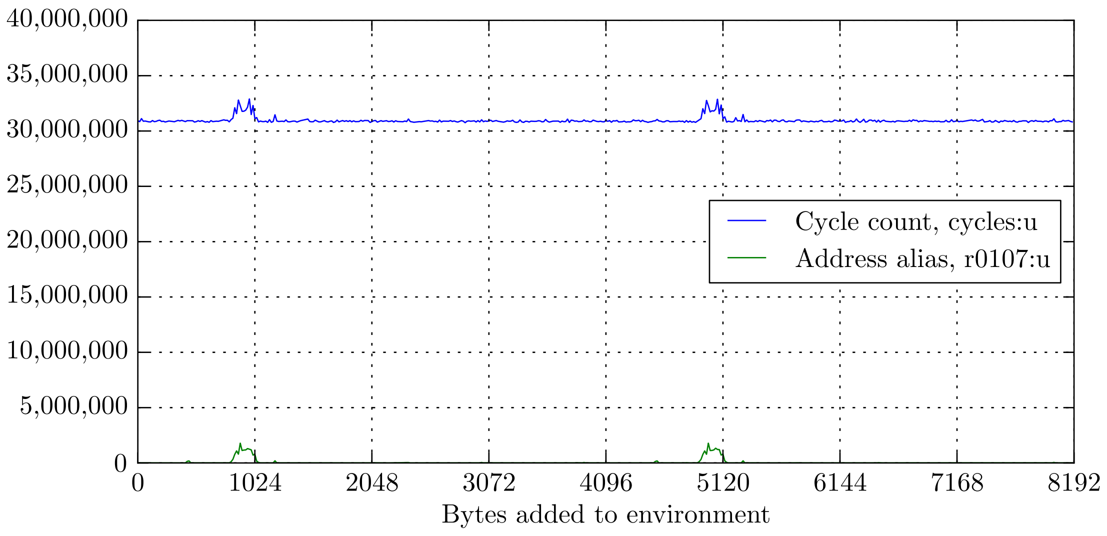

Figure 5.5: Performance counter statistics after stack alignment fix is added to apply_extra_iter, almost all bias removed.

Table 5.1: Cycle count statistics over 512 runs, sampling uniformly over two 4 KiB environment size periods. Mean and maximum is improved with forced alignment, while the constant overhead impacts expected (median) performance.

	Min	Median	Mean	Max
Default version	30,337,010	30,445,740	31,051,690	37,349,700
Kernel modification	30,760,320	30,885,300	30,947,460	32,909,300

Evaluating the Worst Case Scenario

With address randomization disabled, both stack and heap locations will be fixed for all environment sizes. This is exploited in the kernel-level stack alignment fix we have outlined, which only works when input and output buffers are close to 0x000. If however heap buffers happen to reside on the opposite end of a 4 KiB segment, both stack and heap addresses will be close considering the last 12 bits. By allocating another buffer of 7400 B using malloc before calling fftw_malloc, the input and output arrays are pushed to 0x604**da0** and 0x604**f00** respectively. The relative position between each memory segment in this scenario is shown in Figure5.6.

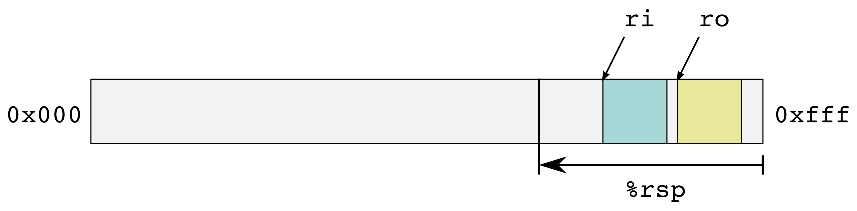

Figure 5.6: Offsetting heap addresses to always overlap with fixed stack pointer.

With both heap and stack addresses suffixes located on the “high” end, there will *always* be aliasing. The effect of this change is shown in Figure 5.7. A constant number of alias events are added across all environment sizes, resulting from the worst case memory alignment.

A Generic Solution

The static rule of aligning stack to page boundary only works with ASLR disabled, and because our fftw_malloc happens to place “ri” and “ro” close to the lower end of a 4 KiB range of address suffixes. A complete solution would require a function at least these three parameters to calculate the optimal stack offset:

$$f\,(\text{input}, \text{output}, \text{stack position}) \rightarrow \text{offset}$$

A plausible solution to the worst case scenario from Figure 5.6 would be to align stack to start at input pointer, no longer overlapping any heap segments.

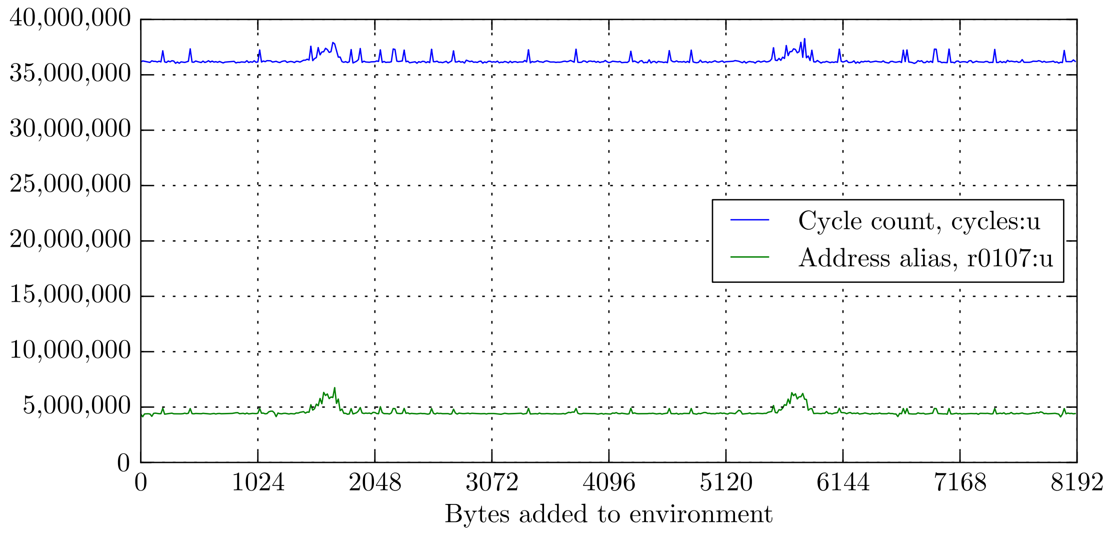

Figure 5.7: Worst case scenario when resetting stack pointer to 0x000. Overlap with heap alignment results in aliasing consistently across all environment sizes.

However, given the somewhat discouraging results from our simple static fix, we did not attempt to implement a function like this for the n1fv_16 kernel. Using inline assembly, the cost of additional dynamic instructions will quickly cancel out any speedup we might get.

5.1.3 Eliminating Bias at the User Level

The ideal solution to aliasing would be a transparent one, where library users do not need to worry about optimal stack alignment. However, when trying to account for aliasing within FFTW itself in the previous section, we found that too many dynamic instructions are added in the process. The reason for this is that the stack correction is implemented inside the hot loop, where fftw_execute is called repeatedly. A possible user level modification would be to align stack *before* the loop, which should add only a negligible overhead. The relevant parts of a modified main function is shown in Listing 5.3.

With ASLR disabled, keeping the heap addresses constant through all runs, we are now able to avoid *all* alias effects for the n1fv_16 kernel. If we were to plot the results, the graph for cycle count and alias events would be completely flat. As shown in Table 5.2, the average cycle count is reduced from 31,051,690 to 30,425,240, a speedup of 2 %.

Listing 5.3 Modified test program with forced stack alignment in main. Stack pointer is saved to a temporary variable before aligned to a 4 KiB boundary, then restored once we are done.

```
void *rsp;
__asm__ volatile (
    "movq  %%rsp, %0;"
    "andq  $-4096, %%rsp;"
    : "=r"(rsp) : : );

for (int i = 0; i < X; ++i)
    fftw_execute(p);

__asm__ volatile (
    "movq   %0, %%rsp;"
    : : "r"(rsp) : );
```

Table 5.2: Cycle count statistics over 512 runs, sampling uniformly over two 4KiB environment size periods. With stack alignment in main, the program performs equally well for all environment sizes.

	Min	Median	Mean	Max
Default	30,337,010	30,445,740	31,051,690	37,349,700
Kernel modification	30,760,320	30,885,300	30,947,460	32,909,300
User modification	**30,350,120**	**30,427,600**	**30,425,240**	**30,585,470**

5.1.4 Bias in Other Kernels

FFTW consists of a quite large collection of automatically generated codelets. While we have only looked extensively at one particular kernel, similar bias effects can be experienced for other plans and input sizes. We find that alias effects are apparent in all kernels we tried, a selection of them shown in Figure 5.8. Each kernel shows a different pattern, with clear bias towards some environment sizes (stack alignments) that performs noticeably better than others.

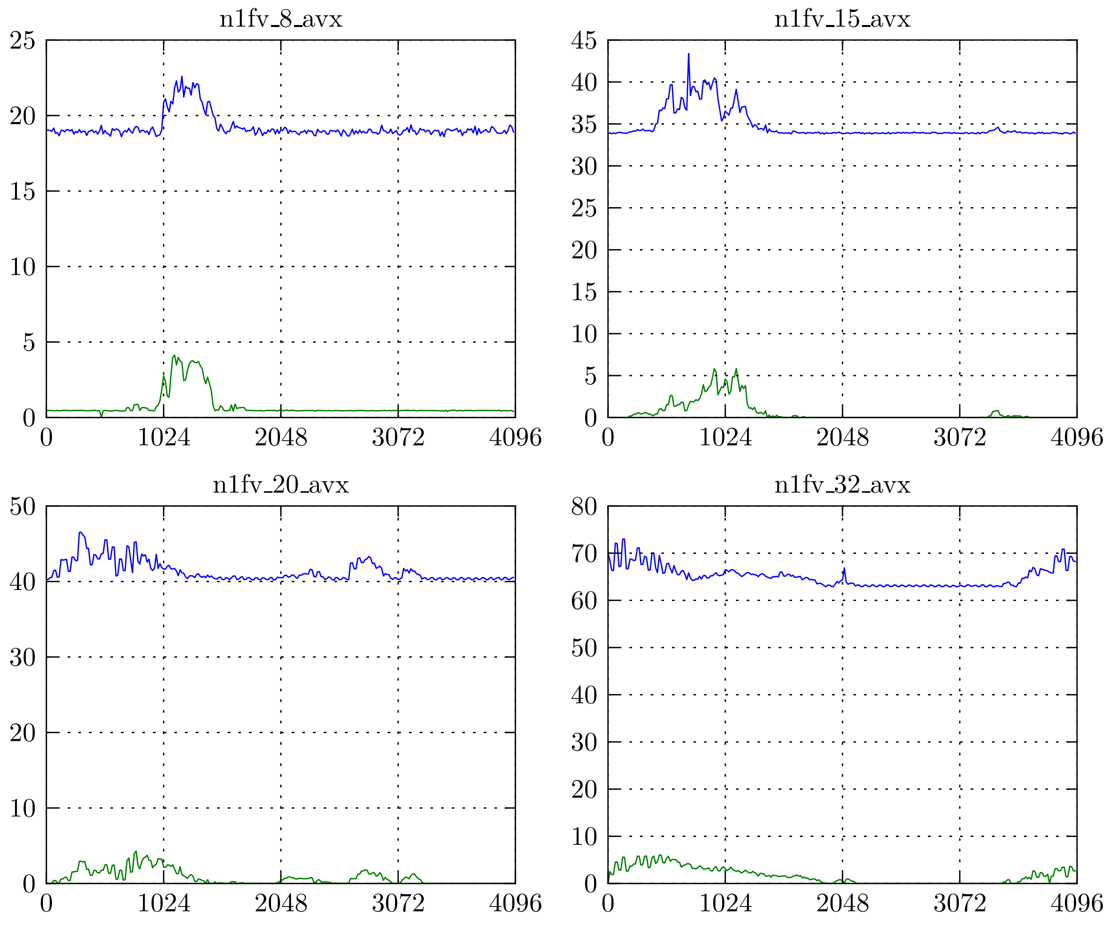

Figure 5.8: Cycle and alias counts for four additional kernels, for 8, 15, 20 and 32 element complex DFTs respectively. Plot shows X = 200,000 iterations of the kernel, incrementing environment size (horizontal axis) between each run, up to 4096 bytes. Vertical axis is given in millions.

5.1.5 Discussion

In this section, we have shown how bias from environment size can affect small kernels in FFTW. The n1fv_16 kernel was studied in detail, showing how aliasing occur from collisions between stack and heap areas. There is clearly a potential for speedup by accounting for aliasing, and we show how manually aligning stack can be a possible solution. With the modification to kernel invocation, we were able to remove almost all of the bias effects. Unfortunately, our solution suffers from overhead of additional dynamic instructions. Having library users consider bias is shown to be feasible, and we prove that there is at least a possibility of 2 % improvement on average for the n1fv_16 kernel. However, requiring users of the library to manually adjust stack before calling fftw_execute is not a satisfactory solution. A more reasonable approach would be to incorporate these ideas in the steps performed *prior* to executing a plan.

Planning for Alias A realistic solution for handling bias is to give this responsibility to the planner. Note that the heap addresses of input and output are already required when computing a plan.

```
fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
```

As we have shown, the optimal stack position for each kernel depends on the memory addresses of input and output arrays, but the component that is missing is stack position. In principle, planning using a more thorough planner which considers empirical timings (for example using `FFTW_MEASURE`) could cancel out bias if the following were true:

1. Multiple alternative kernels with varying alias characteristics exists.
2. The call chain is the same during benchmarking in the planner as when the plan is later executed. Stack addresses during planning must be the same.

The planner can not know the stack depth of which a user calls fftw_execute with a given plan. Bias could however be handled, if the plan considered stack alignment at the time of execution. Alternative code paths after calling fftw_execute could be chosen depending on the current stack position.

Allocating Alias-free FFTW comes with fftw_malloc, which is a wrapper around regular malloc with stronger alignment guarantees. The purpose of using this is to optimize for SIMD instructions. In cases where address conflicts between heap data can occur, extending fftw_malloc to avoid returning aliased heap segments would be a realistic solution. For the n1fv_16 kernel, we found that alias in heap addresses did not matter. However, this is not necessarily true for all kernels. Looking into possible improvements here might be worthwhile.

Compiler Code Generation Our attempts at aligning stack at the kernel level introduced overhead from adding inline assembly. The compiler already does some stack alignment operations, which we more or less discard by overwriting the stack pointer. An alternative solution could be to give the responsibility to the compiler. All the information necessary is available at compile time; the compiler knows every instruction and memory access, and can in principle predict where aliasing might occur. The compiler could then implement dynamic stack alignment based on considering both heap buffers and current stack pointer – like the "generic solution" that was previously discussed.

Conclusions

Despite being highly optimized and automatically tuned, kernels in FFTW still suffers from bias effects. The auto tuning mechanism is designed to deal with architecture specific optimizations, but it fails to account for aliasing stack accesses. Further work on address aliasing issues in FFTW could lead to real performance improvements.

5.2 ATLAS

For our second case study, we will look at alias effects in ATLAS[4] (Automatically Tuned Linear Algebra Software) [28]. ATLAS is an implementation of the BLAS API[4] for linear algebra routines, and a critical component in many high performance applications.

BLAS Basic Linear Algebra Subroutines (BLAS), is the de facto standard API for high performance linear algebra routines[5]. The functionality is divided into three categories:

Level 1 Scalar and vector operations, such as dot product and vector addition.

Level 2 Matrix-vector operations, such as `gemv` for general matrix-vector multiplication.

Level 3 Matrix-matrix operations, including the widely applied `gemm` routine for general matrix-matrix multiplication.

Many highly optimized implementations of BLAS exists, ATLAS being a widely used and open source alternative. One of the key features of ATLAS is that it uses automatic tuning to optimize for cache efficiency [27].

[4]Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/
[5]BLAS (Basic Linear Algebra Subprograms) http://www.netlib.org/blas/

Installation and Configuration We use the currently latest version 3.10.1 of ATLAS, built as a shared library from source. The automatic tuning happens during the build process. A series of test programs are run to determine cache edges and other properties of the hardware, which in turn affects the resulting binary.

Methodology The experimental setup and methodology described in Chapter 3 is applied. Carefully monitoring cache is particularly important in this case study, as BLAS performance heavily relies on cache efficiency. Relevant performance statistics for cache hit ratios will be considered to rule out cache as the cause of any bias effects.

Potential for Bias The idea is to reproduce similar scenarios to what we saw in Section 4.1.2, where aliasing was caused by linear accesses to pairs of aligned heap allocated data. Functions operating on vectors, from BLAS Level 1 or 2, seems most likely to have potential for similar characteristics. Using heap allocated memory, address aliasing is an artifact of the allocator used. With different versions or configurations of memory allocators, any performance impact from aliasing can qualify as measurement bias. In the following sections, we will study bias from address aliasing in the Level 2 function gemv specifically, which computes matrix-vector multiplication.

5.2.1 Address Aliasing in Matrix-Vector Multiplication

Consider matrix-vector multiplication of the form $\boldsymbol{y} = A\boldsymbol{x}$. Let A be of size $M \times N$, where M is the number of rows.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & & a_{0,N} \\ a_{1,0} & & & \\ & & \ddots & \\ a_{M,0} & & & a_{M,N} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_M \end{bmatrix}$$

The corresponding BLAS function is the level 2 `gemv` routine, computing the more general matrix-vector product given as

$$\boldsymbol{y} = \alpha \mathrm{op}\,(A)\,\boldsymbol{x} + \beta \boldsymbol{y}$$

Here, α and β are constants, and $\mathrm{op}\,(A)$ is an optional transpose or complex conjugate of the matrix. A typical invocation of this routine is shown in Listing 5.4. We set $\alpha = 1$, $\beta = 0$ and $\mathrm{op}\,(A) = A$ to reduce the formula to $\boldsymbol{y} = A\boldsymbol{x}$. Note that A is declared as `CblasColMajor`, meaning we impose a column major ordering of the data. The prefix indicates data type, in this case d for double precision.

Benchmarking this program with matrix dimensions $N = M = 8192$ and iteration count $K = 1$, we see that performance varies quite significantly with different

Listing 5.4 Computing double precision matrix-vector multiplication $\boldsymbol{y} = A\boldsymbol{x}$ using `cblas_dgemv`

```
int main() {
    const double alpha = 1.0, beta = 0.0;

    double *A = malloc(sizeof(double) * M * N);
    double *x = malloc(sizeof(double) * N);
    double *y = malloc(sizeof(double) * M);

    for (int i = 0; i < K; ++i)
        cblas_dgemv(CblasColMajor, CblasNoTrans, M, N, alpha,
            A, M, x, 1, beta, y, 1);
    return 0:
}
```

Table 5.3: Performance counter statistics for a simple program invoking ATLAS' cblas_dgemv with matrix size 8192 × 8192. Cycle count and aliasing varies with different heap addresses for A, $\boldsymbol{x}$ and $\boldsymbol{y}$.

	&A	&x	&y	Cycles (cycles:u)	Alias (r0107:u)
(a)	0x2aaaac292**010**	0x607**010**	0x617**020**	195,155,171	34,095,566
(b)	0x2aaaac292**010**	0x607**010**	0x6170b0	186,788,588	5,724,675
(c)	0x2aaaac292010	0x6073d0	0x6173e0	180,778,001	37,555

heap addresses of A, $\boldsymbol{x}$ and $\boldsymbol{y}$. The performance counter statistics for three different address configurations is shown in Table 5.3. As might be suspected, the worst case seems to be when all memory buffers align closely on the same 12 address bit suffix.

Because of overhead from calls to malloc making large heap allocations, these measurements do not represent the true cost of aliasing in dgemv. To better be able to assess the effects of aliasing, the call to gemv needs to be isolated. We use the program from Listing 5.4 to make two benchmarks, one with iteration count $K = 1$ and another with $K = 101$. The approximated cost for a single invocation of dgemv can be expressed as

$$t_{\text{estimate}} = \frac{t_{K=101} - t_{K=1}}{100}$$

where t represents some metric, such as the number of cycles. Subtracting the $K = 1$ run removes the constant overhead from the $K = 101$ run. Dividing by 100 averages the values from the remaining iterations. Other iteration counts could have been used as well.

Table 5.4: Estimated cost of a single invocation of `cblas_dgemv` with matrix size 8192×8192 in ATLAS. Each column shows statistics for different heap addresses. The 12 bit address suffixes for A, $\boldsymbol{x}$ and $\boldsymbol{y}$ in each case is (0x010, 0x010, 0x020) for column a, (0x010, 0x010, 0xb0) for b and (0x010, 0x3d0, 0x3e0) for c.

	(a)	(b)	(c)
UnHalted Core Cycles	60,578,750	51,654,310	46,059,028
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	**33,444,229**	**3,805,742**	**2,322**
MEM_UOPS_RETIRED.STORES	16,781,370	16,781,370	16,781,370
MEM_UOPS_RETIRED.LOADS	58,745,150	58,745,734	58,745,216
MEM_LOAD_UOPS_RETIRED.HIT_LFB	6,106,659	7,942,348	6,043,806
MEM_LOAD_UOPS.RETIRED.L1_HIT	52,189,609	49,964,781	51,675,427
MEM_LOAD_UOPS.RETIRED.L2_HIT	448,620	838,025	1,025,608
MEM_LOAD_UOPS.RETIRED.LLC_HIT	-29	531	178
MEM_LOAD_UOPS.RETIRED.LLC_MISS	7	61	18
LOAD_HIT_PRE.HW_PF	14,425,267	13,428,234	6,332,654
CYCLE_ACTIVITY.CYCLES_LDM_PENDING	121,057,180	103,250,874	91,998,759

Estimated performance counter statistics for each of the three heap address configurations are shown in Table 5.4. In addition to cycle count and alias events, a number of relevant metrics related to cache activity are also included. Notice that almost all load micro-ops are served by either the line fill buffer or L1 cache in all cases. Only a small amount of loads come from L2, and almost none from L3. The hit rate for L1 actually decreases somewhat (considering LFB as well) with better execution time. This could be explained by less time for prefetchers to feed the L1 cache with data. The hardware prefetch counter indicates more hits in cases a and b. Again, we find that cache efficiency does not explain the performance cliffs we observe.

Our results show that address aliasing between matrix and vector heap buffers can significantly impact performance of dgemv in ATLAS. Variations in heap addresses alone can give a speedup of more than 31 %.

5.2.2 Dealing With Aliasing

For the particular case we investigated, a good heuristic is to align heap segments “far apart” within the 4 KiB area of 12 bit suffixes. More specifically, it appears that address suffixes of A and $\boldsymbol{y}$ are the most important to separate.

As described in Section 4.1.2, aliasing cases like these can be accounted for in software using *padding* techniques. A possible run time solution to adjust heap addresses can be realized as follows:

1. Allocate some extra space for one of the vectors when calling malloc, for instance `sizeof(double) * (M + 0x100)` for $\boldsymbol{y}$.

2. Check the returned pointers for potential alias, i.e. the difference between `&A` and `&y`. Offset using pointer arithmetic into the array with extra padding at the end, i.e. `y += 0x100`.

Another option is to explicitly account for "worst cases" in the implementation of routines that are vulnerable to aliasing. Addresses can be explicitly checked for potential conflicts in cblas_dgemv, and if possible branch to code that will not suffer from aliasing.

5.2.3 Discussion

We briefly looked at potential bias effects in linear algebra routines, and found that matrix-vector multiplication in ATLAS suffered from heap address aliasing. Close alignment within a 4K segment of address suffixes can have a significant performance penalty, with more than 31 % difference between the worst and best case. It becomes *necessary* to consider effects of address aliasing when optimizing programs that rely on ATLAS' implementation of dgemv.

The extent of aliasing issues within other function in ATLAS, or among other BLAS implementations, is still an open question. Because of time limitations, we were not able to investigate other potential aliasing cases. However, it seems very *likely* that similar problems are prevalent for linear algebra implementations in general. There is probably a significant potential for speedup in many BLAS implementations by actively avoiding bad address alignment. Our results should motivate further work to address these issues.

Chapter 6

Conclusions and Future Work

In this thesis, we have studied effects of contextual bias on the Intel Core "Ivy Bridge" architecture, and shown how avoiding bias can also improve performance. The premise for our work relies heavily in previous work done by Mytkowicz et al. Their study of measurement bias in earlier architectures showed that changes to environment variables and link order could heavily impact performance. Furthermore, the effects were concluded to be unpredictable and difficult to account for. Questions of what actually causes bias are left largely unanswered, which leaves room for valuable research. Our goal was to gain a better understanding of the mechanisms causing bias effects, and if possible use this knowledge to improve performance of real world applications.

As a starting point for our research, we chose to study the micro-kernel first presented in [17]. It was not clear whether bias effects were even an issue on our architecture, as previous work studied mostly Core 2 and earlier processors. Being able to reproduce similar effects from changing environment size motivated continued exploration. In Chapter 4, we presented an in-depth study of what was causing bias for that particular program. Through careful measurements using hardware performance counters, we found that address aliasing between local and static variables could explain what was happening.

With an understanding of the possible performance penalties incurred from aliasing, we explored what other use cases might be affected by this. The fact that malloc often allocates on page boundary by default proved to be troubling, essentially guaranteeing bad performance from aliasing in many cases. The convolution example exploited this synergy between heap allocation and 4K aliasing, showing that some knowledge of the hardware could provide massive speedups.

The Loop Stream Detector had been previously mentioned as a possible cause of bias. Studying the LSD proved to be challenging, because most of the documentation was either incomplete or misleading. Using several hand coded assembly programs, we were able to identify the correct parameters for the Ivy Bridge LSD. Interestingly, the capacity was improved compared to previous generations, not reflected in the documentation. With a clear understanding of the chunk fetch limit, we were able to identify cases where link ordering caused serious bias effects. The underlying reason proved to be code layout with respect to 32 byte chunks, where a maximum of 12 chunks would fit in the LSD cache.

In our case studies, we chose to concentrate on highly optimized numerical libraries. The reasoning behind this was that these kind of applications are more likely to incorporate extremely processor specific optimizations in order to achieve the maximum possible performance. We chose to focus on address aliasing effects as opposed to the Loop Stream Detector when looking for instances of bias. Compared to altering code layout by changing link order, it was more convenient to experiment with changes to the environment size or heap alignment for large applications. We were able to identify bias towards certain stack positions for small kernels in FFTW, caused by 4K aliasing. Attempts were made to align stack dynamically in the kernels, but the added dynamic instructions quickly incurred too large of an overhead to be worth it.

Lastly, we wanted to investigate whether BLAS libraries also suffered from bias. We suspected that certain operations might be sensitive to aliasing between heap buffers, similar to the effects pointed out in Chapter 4. The impact of address aliasing in matrix-vector multiplication proved to be huge, even for an automatically tuned ATLAS installation. By manually adjusting alignment of matrix and vector buffers, speedups of more than 31 % could be achieved for invocations of dgemv.

6.1 Contributions

Previous work identified *environment variables* and *link order* as potential triggers of bias effects in performance analysis [17]. The main purpose of this work was to gain an understanding of the underlying effects causing measurement bias. We have identified and documented two such architecture-specific properties of the Intel Core "Ivy Bridge":

- 4K address aliasing: Alias on the last 12 bits can cause false conflicts in the memory ordering system, biasing performance towards certain data layouts.
- Loop Stream Detector: Limitations in micro-op caches favors code with certain static memory layouts.

It is important to point out that limitations of the LSD and the existence of 4K aliasing is not new or unknown. Although scarcely documented, both of these issues are discussed to some extent in the processor vendor manuals. However, modern CPU architectures are tremendously complicated, and very few people would even care to read the manuals for arcane details of these topics.

Our work extends to the documented processor behavior, and links these properties to the concept of measurement bias. We show how code and data layout in memory can affect program performance, finally providing satisfactory explanations of measurement bias that can be observed from changing link order or environment variables. We also present other bias triggers not mentioned in previous work. Functionality of heap allocators is important for address aliasing, and bias effects can occur from using different versions or implementations. In addition to link ordering, other properties that changes code layout can affect usage of the Loop Stream Detector. Things like the order of functions within a source file, or the length of external symbols, can be significant.

Unlike stated in previous work, we found that bias is in fact *not* entirely unpredictable. Furthermore, a clear understanding of what causes bias is necessary in order achieve optimal performance in real applications. Our results are relevant to anyone who care about optimizing program performance on modern Intel architectures.

6.2 Directions for Future Work

Studying sources of bias effects proved to be complicated and challenging. One of the main problems was lack of accurate documentation, making the processor a kind of “black box”. We believe that we have only scratched the surface when it comes to interesting and lesser known architectural features. The optimization manual [10] is a good starting point for further research, describing how various hardware features interact and operate. Some of the features we think might be worthwhile to investigate include;

- Decoded ICache limitations: We observed corner cases of the Loop Stream Detector that were probably related to micro-ops not being cached. Figuring out the exact parameters for the micro-op cache could help compilers generate optimal code layout.
- Hyper-threading: This feature was disabled throughout our testing to avoid any interference. We imagine there are many interesting performance implication occurring from threads competing about resources. For example, the LSD micro-op limitation being shared between logical cores is clearly a potential source of bias towards systems without hyper-threading enabled.

Further work is needed to lift more of the knowledge buried among technical details in official manuals. With increased awareness of these effects, there is clearly a potential for improving performance of existing programs.

6.2.1 Compiler Optimizations

The most natural place to encode architectural-specific knowledge used for optimization is of course in compilers. In an ideal world, programmers would not need to care about intricate corner cases in hardware to achieve the best performance.

Optimizing code layout for maximal usage of the Loop Stream Detector could be a worthwhile endeavor. We did not investigate to what degree this is handled by modern compilers other than GCC, but would not be surprised if there is considerable room for improvement.

Generating alias-free code could potentially also be done by the compiler. We show how aliasing can be avoided by adding alternative code paths in Chapter 4, which at least in principle shows that compilers could handle this in some cases. Code that relies on heap-allocated buffers seems like a particularly difficult thing to optimize for. A starting point could be to detect functions that might suffer from aliasing via some static analysis, perhaps issuing a warning.

Using *profiling* information could be another way of identifying potential bias related issues. GCC has support for two-stage compilation, using profiling information from an instrumented version of the program to aid in optimization of the final compilation. Encoding knowledge of for example 4K aliasing in a profiling step seems feasible.

6.2.2 Alias-free Allocators

We show that typical memory allocators are prone to cause aliasing in *all* cases, because heap memory is often page aligned. As far as we know, there are no malloc implementations that specifically handles aliasing issues. Developing good heuristics on the optimal heap addresses to return from malloc would be an interesting topic for further research. An obvious improvement would be to perturb addresses of allocations done with mmap, avoiding identical 12 bit address suffixes of large allocations.

6.2.3 Library Optimizations

In our case studies, we found that bias effects from address aliasing affected both FFTW and ATLAS. For highly optimized numerical libraries in particular, there should be incentive to account for address aliasing to improve performance.

In cases where aliasing depend on heap allocated memory, a possible solution could be to provide special-purpose wrappers around malloc. FFTW already has fftw_malloc, which would be a perfect place to encode knowledge about aliasing. We did not find any effects from conflicts between heap areas in the particular kernel we studied, but other kernels or code within FFTW might be sensitive to this. Providing a similar function for BLAS implementations is also an option.

There is probably also opportunities for writing code that is less sensitive to data alignment. In the case of matrix-vector multiplication, it might be possible to access data in a different order depending on heap addresses. By explicitly checking the addresses of input parameters, different code paths can be chosen to avoid aliasing. With potential for more than 31 % performance improvement in the worst case, the added cost and complexity is probably acceptable.

The extent and impact of bias in other libraries is also an open question. Of the two we studied, we were able to identify address aliasing fairly easily. We suspect that this is a common phenomenon, and worth investigating for developers of performance critical applications.

6.2.4 Other Architectures

We limited our study to one specific architecture, the at the time of writing most recent "Ivy Bridge" from Intel. An interesting question is how effects from Loop Stream Detector limitations and 4K aliasing translate to other CPU architectures. Effects from 4K aliasing are documented in architectures as far back as the first Core processors [10]. Limitations of the LSD have changed throughout the last several generations, suggesting that the particular examples we present might not behave similarly on other Intel processors. Processor architectures change at a rapid pace, and by the time this thesis is published the "Ivy Bridge" will have been replaced by "Haswell". More research is needed on sources of bias effects, both in previous and future architectures.

6.3 Final Words

Our work shows that 4K aliasing and the Loop Stream Detector, both intricate and highly architecture-specific properties of the processor, can have significant performance impact in real software. Raising awareness of the importance of memory layout of code and data is relevant to both users and developers of performance critical software. A continual effort to expand and update this knowledge for current and future architectures is needed.

Bibliography

[1] Reference for processor events. Available from `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/win/ug_docs/reference`.

[2] System V Application Binary Interface, Edition 4.1. Available from `http://www.sco.com/developers/devspecs/gabi41.pdf`, March 1997.

[3] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not. 35*, 11 (November 2000), 117–128.

[4] Blackford, L. S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., and Whaley, R. C. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw. 28*, 2 (2002), 135–151.

[5] Doweck, J. Inside Intel® Core™ microarchitecture. Presentation, available from `http://www.hotchips.org/wp-content/uploads/hc_archives/hc18/3_Tues/HC18.S9/HC18.S9T4.pdf`.

[6] Doweck, J. White paper: Inside Intel® Core™ microarchitecture and smart memory access, 2006.

[7] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. In *Proceedings of the IEEE* (2005), vol. 93, pp. 216–231.

[8] Hundt, R., Raman, E., Thuresson, M., and Vachharajani, N. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2011), CGO '11, IEEE Computer Society, pp. 1–10.

[9] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering 9*, 3 (2007), 90–95.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, March 2013.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, January 2013.

[13] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[14] Knights, D., Mytkowicz, T., Sweeney, P. F., Mozer, M. C., and Diwan, A. Blind optimization for exploiting hardware features. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (Berlin, Heidelberg, 2009), CC '09, Springer-Verlag, pp. 251–265.

[15] Lomont, C. Introduction to x64 assembly. Available from `http://software.intel.com/en-us/articles/introduction-to-x64-assembly`, 2012.

[16] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. We have it easy, but do we have it right? In *NSF Next Generation Systems Workshop* (2008), pp. 1–5.

[17] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 265–276.

[18] Mytkowicz, T., Sweeney, P. F., Hauswirth, M., and Diwan, A. Observer effect and measurement bias in performance analysis, 2008.

[19] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design – The Hardware / Software Interface (Revised 4th Edition).* The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[20] Peri, R. Performance Monitoring on Intel® Core™ i7 Processors. Available from `http://cscads.rice.edu/workshops/summer09/slides/performance-tools/CSCADS_NHM_PMU.pdf`, 2009.

[21] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307.

[22] Smith, M. D. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization* (New York, NY, USA, 2000), DYNAMO '00, ACM, pp. 1–11.

[23] Tanenbaum, A. S. *Modern Operating Systems (3. ed.).* Pearson Education, 2008.

[24] Vuduc, R., and Demmel, J. W. Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW. In *Semantics, Application, and Implementation of Program Generation* (2000), vol. 1924, Springer Berlin Heidelberg, pp. 190–211.

[25] Walpole, R. E., Myers, R. H., and Myers, S. L. *Probability & Statistics for Engineers & Scientists (9th Edition).* Prentice Hall, 2011.

[26] Wechsler, O. Inside Intel® Core™ microarchitecture: Setting new standards for energy-efficient performance. *Technology@Intel Magazine* (2006).

[27] Whaley, C., Petitet, A., and Dongarra, J. J. Automated Empirical Optimization of Software and the ATLAS Project. In *Parallel Computing* (2000), vol. 27.

[28] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 1998), Supercomputing '98, IEEE Computer Society, pp. 1–27.

Appendix A

List of Performance Counters

There are two categories of performance events; *architectural events* are defined across families of microarchitectures, where the same event codes are supported on a range of processors. *Non-architectural* events are specific to each architecture. The following includes a reference of important counters discussed in this thesis.

Architectural Events

The following architectural events are supported on processors based on the Intel Core microarchitecture. Adapted from Table 19-1 in the Software Developer's Manual [12].

Event num.	**Umask value**	**Event Mask Mnemonic**	**Perf mnemonic**	**Description**
0x3C	0x00	UnHalted Core Cycles	cycles	
0x3C	0x01	UnHalted Reference Cycles	bus-cycles	
0xC0	0x00	Instruction Retired	instructions	
0x2E	0x4F	LLC Reference	cache-references	Last level cache references
0x2E	0x41	LLC Misses	cache-misses	Last level cache misses
0xC4	0x00	Branch Instruction Retired	branches	Branch instruction at retirement
0xC5	0x00	Branch Misses Retired	branch-misses	Mispredicted Branch Instruction at retirement

Non-Architectural Events

This following table contains a subset of the non-architectural performance counters available on 3rd generation Intel Core "Ivy Bridge" processors. The official reference can be found in Table 19-5 in the Software Developer's Manual [12]. Note that LSD_OVERFLOW and LSD.UOPS are included in this list, despite not being officially supported.

Event num	**Umask value**	**Event Mask Mnemonic**	**Description**
0x03	0x02	LD_BLOCKS. STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded
0x07	0x01	LD_BLOCKS_PARTIAL. ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address
0x20	0x01	LSD_OVERFLOW	Counts number of loops that can't stream from the instruction queue
0x2E	0x4F	LONGEST_LAT_CACHE. REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.
0x2E	0x41	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.
0x3C	0x00	CPU_CLK_UNHALTED. THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.
0x3C	0x01	CPU_CLK_THREAD_UN-HALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.
0x4C	0x02	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.
0x5C	0x01	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0
0x5C	0x02	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0

0xA1	0x01	UOPS_DISPATCHED_PORT. PORT_0	Cycles which a uop is dispatched on port 0
0xA1	0x02	UOPS_DISPATCHED_PORT. PORT_1	Cycles which a uop is dispatched on port 1
0xA1	0x0C	UOPS_DISPATCHED_PORT. PORT_ 2	Cycles which a uop is dispatched on port 2
0xA1	0x30	UOPS_DISPATCHED_PORT. PORT_ 3	Cycles which a uop is dispatched on port 3
0xA1	0x40	UOPS_DISPATCHED_PORT. PORT_ 4	Cycles which a uop is dispatched on port 4
0xA1	0x80	UOPS_DISPATCHED_PORT. PORT_ 5	Cycles which a uop is dispatched on port 5
0xA2	0x01	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason
0xA2	0x04	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available
0xA3	0x01	CYCLE_ACTIVITY. CYCLES_L2_PENDING	Cycles with pending L2 miss loads
0xA3	0x02	CYCLE_ACTIVITY. CYCLES_LDM_PENDING	Cycles with pending memory loads
0xA3	0x08	CYCLE_ACTIVITY. CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads
0xA3	0x04	CYCLE_ACTIVITY. CYCLES_NO_EXECUTE	Cycles of dispatch stalls
0xA8	0x01	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector
0xC0	0x00	INST_RETIRED.ANY_P	Number of instructions at retirement.
0xC4	0x00	BR_INST_RETIRED. ALL_BRANCHES	Branch instructions at retirement.
0xC5	0x00	BR_MISP_RETIRED. ALL_BRANCHES	Mispredicted branch instructions at retirement.
0xD0	0x01	MEM_UOPS_RETIRED. LOADS	Qualify retired memory uops that are loads. Combine with umask 0x10, 0x20, 0x40, 0x80.

0xD0	0x02	MEM_UOPS_RETIRED. STORES	Qualify retired memory uops that are stores. Combine with umask 0x10, 0x20, 0x40, 0x80.
0xD0	0x80	MEM_UOPS_RETIRED. ALL	Qualify any retired memory uops. Must combine with umask 0x01, 0x02, to produce counts.
0xD1	0x01	MEM_UOPS_RETIRED. L1_HIT	Retired load uops with L1 cache hits as data sources.
0xD1	0x02	MEM_UOPS_RETIRED. L2_HIT	Retired load uops with L2 cache hits as data sources.
0xD1	0x04	MEM_UOPS_RETIRED. LLC_HIT	Retired load uops whose data source was LLC hit with no snoop required.
0xD1	0x20	MEM_UOPS_RETIRED. LLC_MISS	Retired load uops whose data source is LLC miss
0xD1	0x40	MEM_UOPS_RETIRED. HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.

Appendix B

Benchmark Script

A Python script is used as a wrapper around `perf stat` to collect performance counters under varying execution contexts. A dummy variable “FOO” is used to incrementally add bytes to the environment. The command line interface is similar to `perf`, with the same syntax for specifying which events to monitor. The following command gathers cycles:u and r0107:u (address alias) over 256 runs of “loop”, while incrementing the environment by 16 bytes each time.

```
$ ./lperf ./loop -e cycles:u,r0107:u -n 256 --environment
   -increment 16
```

The script relies on a file containing event codes and mnemonics, by default called “counters”. Output is exported as comma separated values to “stat.csv”.

```
#!/usr/bin/env python
import subprocess, argparse, sys, copy
from scipy import stats

# Use at most 4 counters simultaneously
n_counters = 4

def disable_layout_randomization():
    subprocess.call('sudo bash -c "echo 0 > /proc/sys/kernel/
       randomize_va_space"', shell=True)

def benchmark(events, args):
    for e in events:
        e['count'] = [0]*args.num
        e['variance'] = [0]*args.num

    environment = {'FOO':'0'*args.environment_offset}
    argument = args.argument_offset;

    for run in range(args.num):
        tempfile = 'stat.tmp.' + str(run) + '.dat'
        subprocess.call('cp /dev/null '+tempfile, shell=True)
        for batch in [events[i:i+n_counters] for i in range(0, len(
           events), n_counters)]:
            e = ','.join(map(lambda x: x['code'], batch))
            c = ' '.join(['perf stat -r', str(args.repeat), '-x","'
                , '-e', e, args.program, str(argument), '0>>',
                tempfile])
            p = subprocess.Popen(c, env=environment, shell=True)
            p.wait()

        with open(tempfile, 'r') as f:
            for i in range(len(events)):
                line = f.readline().strip().split(',')
                if (line[0] == "<not counted>"):
                    continue
                events[i]['count'][run] = float(line[0])
                if args.repeat > 1:
                    events[i]['variance'] = float(line[2][:-1])
        subprocess.call('rm '+tempfile, shell=True)

        environment['FOO'] += '0'*args.environment_increment
        argument += args.argument_increment;

    return events

def correlation(events, reference_event):
    reference = events[0]
    for i in range(len(events)):
        if events[i]['code'] == reference_event or events[i]['
           perfmn'] == reference_event:
            reference = events[i]
            break
    for e in events:
        e['pearson'], _ = stats.pearsonr(reference['count'], e['
           count'])

def export_csv(events, filename):
    with open(filename, 'w') as f:
        for line in events:
            row = [line['mnemonic'], line['code']] + [line['pearson
                ']] + map(str, line['count'])
            f.write(','.join(map(str, row)))
            f.write('\n')

# Read file containing performance event information
def read_file_events(filename):
    events = []
    with open(filename) as f:
        for line in f:
            code, perfmn, name = map(lambda s : s.strip(), line.
                split('\t'))
            code = ''.join(['r', code.lower(), ':u']) if perfmn ==
                '' else perfmn+':u'
            events.append({'code': code, 'perfmn': perfmn, '
                mnemonic': name})
    return events

# Match events specified with "-e" flag with metadata from file
def filter_events(file_events, include):
    if include == []:
        return file_events
    events = []
    for e in include:
        found = False
        for s in file_events:
            if s['code'] == e or s['mnemonic'] == e:
                found = True
                events.append(copy.copy(s))
        if not found:
            print "Adding unknown event", e
            events.append({'code': e, 'perfmn': '', 'mnemonic': ''
                })
    return events

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='perf runner')
    parser.add_argument('program', help='program, ex ./a.out')
    parser.add_argument('-e', '--events')
    parser.add_argument('-f', '--event-file', default='counters')
    parser.add_argument('-o', '--output', default='stat.csv')
    parser.add_argument('-n', '--num', type=int, default=2)
    parser.add_argument('-r', '--repeat', type=int, default=1)
    parser.add_argument('-c', '--correlate', default='cycles:u')
    parser.add_argument('--environment-offset', type=int, default
       =0)
    parser.add_argument('--environment-increment', type=int,
       default=1)
    parser.add_argument('--argument-offset', type=int, default=0)
    parser.add_argument('--argument-increment', type=int, default
       =0)
    args = parser.parse_args()

    # Make memory layout deterministic
    disable_layout_randomization()

    # List of event properties [[code, perfmnemonic, description]]
    events = filter_events(read_file_events(args.event_file), [] if
         args.events == None else args.events.strip().lower().split(
        ','))
    events = benchmark(events, args)

    # Export result
    correlation(events, args.correlate)
    export_csv(events, args.output)
```