SHIM and Its Applications

Xi Yang

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

November 2019

Except where otherwise indicated, this thesis is my own original work.

Xi Yang
1 November 2019

to Liang and Jayne.

Acknowledgments

I am deeply indebted to my supervisors, Professors Steve Blackburn and Kathryn S. McKinley. They not only patiently supported me in addressing fundamental research problems. But more importantly, they showed me how to be a great teacher, a great parent, a great spouse, and a great humane person.

I am also greatly indebted to my hero, Huailin Chen, whose articles taught me the principles of computer systems and attracted me to the wonderful world of research and engineering.

Many people and organizations have provided generous support and assistance. I would like to thank Dan Luu and Martin Maas for promoting my research, and my managers, Michael Agnich, Charles Tripp, and Xin Xiang, for supporting me in finishing my thesis when I was working full time at TerrainData and Confluent. I also would like to thank my friends in our research lab, Yi, Kunshan, Tiejun, Ting, Ivan, Rifat, Vivek, and Luke, for their collaboration, support, and company. I would like to thank ANU and Google for providing scholarships, Microsoft Research for providing internships, and Centrelink for providing family benefits after my daughter was born

Lastly, I would like to thank my family for their love and support. For my parents, Yangai and Qiudong, who gave me considerable freedom when I grew up and supported me in pursuing my research. For my daughter, Jayne, whose regular sleep schedule provided precious working hours. And, most of all, for my loving wife, my best friend, and also my life mentor, Liang, whose constant support and invaluable life suggestions encouraged me. Now it is my turn to support her in pursuing her remarkable research.

Abstract

Profiling is the most popular approach to diagnosing performance problems of computer systems. Profiling records run-time system behavior by monitoring software and hardware events either exhaustively or—because of high costs and strong observer effects—periodically. Sampling rates thus determine visibility: the higher the sample rates, the finer-grain behavior observable, and thus the better profilers can help developers analyze and address performance problems.

Unfortunately, the sample rates of current profilers are extremely low because of the perturbations generated by their sampling mechanisms. Consequently, current profilers cannot observe insightful fine-grain system behavior. Despite the gigahertz speeds of modern processors, sampling frequencies have been at a standstill—between 1 KHz and 100 KHz—to limit perturbation. This million-cycle gap between two sequential samples blinds profilers to fine-grain behaviors, thus missing root causes of performance problems and potential solutions.

> My thesis is that by exploiting existing underutilized multicore hardware the sample rates of profilers can be increased by orders of magnitude, leading to new profiling approaches, new discoveries of insightful behavior, and new optimizations.

The insights and contributions of this thesis are: 1) We view computer systems as high-frequency signal generators. The high-frequency hardware and software signals that reflect fine-grain system behavior are observable in signal channels: performance counters and shared memory locations. We introduce Shim, a new profiling approach that continuously samples signal channels at resolutions as fine as 15 cycles, which is three to five orders of magnitude finer than current sampling approaches. Shim automatically filters out noisy samples to produce high-fidelity signals. 2) Shim's high-frequency profiling enables a new approach to analyzing and controlling fine-grain system behaviors. We design Tailor, a real-time latency controller for latency-critical web services. Tailor uses a Shim-based high-frequency profiler and an application-level network proxy to continuously monitor and promptly act on the system behaviors that are hazardous to request latency. 3) Shim's fine-grain control of system components enables a new class of online profile-guided optimizations. We introduce Elfen, a Shim-based job scheduler that borrows cycles in short idle periods of latency-critical workloads for batch workloads. Elfen improves CPU utilization significantly without interfering with latency-critical requests by monitoring status changes of latency-critical requests with Shim, and taking real-time scheduling actions.

The history of science shows that an order of magnitude or more improvement in measurement fidelity leads to fundamental new discoveries. This thesis fundamen-

tally alters which software and hardware signals are observable on existing systems, and demonstrates that observing these signals stimulates new optimization opportunities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation, we focus on *general purpose sampling-based profilers* that observe both software and hardware behavior by sampling hardware and software events on existing hardware platforms.

Specifically, we addresse the challenge of designing new high-frequency high-fidelity profilers on existing hardware platforms, discovering insightful fine-grain system behaviors with new profilers, and designing a new class of profile-guided optimizations.

1.1 Motivation

In computing, *performance* is measured by the amount of useful work accomplished by computer systems in a fixed period, and *functionality* is measured by what kinds of useful work computer systems can do. Since the first computer was invented, performance improvements have been driving the development of new functionality. Higher system performance enables engineers to explore new ways of designing and using computer systems, new solutions to challenging problems, and more productive ways of crafting computer systems. For example, in the seventies, researchers at Xero PARC made astonishing innovations—the graphical user interface (GUI), Smalltalk, the What-You-See-Is-What-You-Get (WYSIWYG) editor, laser printing, and Ethernet—that together laid the foundations of modern personal computers. Alan Kay, the Smalltalk designer, attributed these innovations to the Xerox Alto, the highly optimized computer they were using, saying "It (Xerox Alto) was a *time machine* to allow individual researchers starting in 1973 to work about 12–15 years in the future they were trying to invent." [Kay, 2019]. Today, fast, small and energy efficient SoC chips make it possible to build smart phones. Massive parallel computing power, provided by GPUs and datacenters, is the heart of AI and machine learning. To advance the functionality of computer systems, we must keep optimizing performance.

Over decades, Moore's law and Dennard scaling have been major forces pushing system performance forward: faster and smaller transistors gave us more powerful computer hardware every year. Today, as both of these forces are reaching their limits, to get better performance, engineers must optimize the full system stack, including the micro-architecture, operating systems, programing languages, core libraries, and

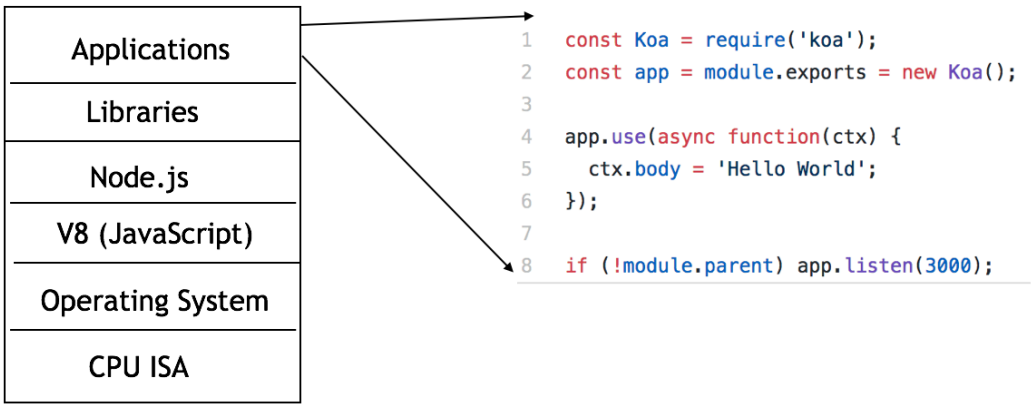


Figure 1.1: Leveraging rich abstractions makes it possible to craft a simple http server in a few lines of JavaScript code.

applications.

Not only does the ending of Moore's law and Dennard scaling force engineers to do full-stack optimizations, but so do the diverse performance requirements of modern computer systems. Today, in addition to latency and bandwidth, performance has many other facets: tail latency, utilization, scalability, energy efficiency, and availability. Many of these performance problems cannot be automatically solved with faster computers, but require engineers to understand root performance problems and design new optimizations.

However, discovering new optimizations is challenging. Just as with other experimental sciences, it requires a scientific discovery process: observing computer systems, understanding system behavior, and designing new optimizations. None of these steps is easy when facing complex computer systems. Butler Lampson quoted Wheeler's words in his Turing lecture, "Any problem in computer science can be solved with another level of indirection." We have been crafting computer systems by leveraging and providing rich abstractions for decades. The result is a complex system stack. The layers of abstractions improve developer productivity significantly. As shown in Figure 1.1, a few lines of JavaScript can implement a simple HTTP server by invoking layers upon layers of libraries, but it also poses the challenge of finding performance problems and discovering optimizations because it is hard to understand system behavior. For example, Dick Sites from Google [Sites, 2015] mentioned that they spent three years on understanding the root causes of some mystery tail requests. After investigating disk traces and analyzing full system stacks across datacenters, they finally found that the root problem was from within the Linux kernel.

Facing complex computer systems, developers need powerful tools to help them observe and analyze fine-grain system behaviors, especially fine-grain interactions between system components. Hardware designers need to understand code patterns of low instruction-per-cycle (IPC) periods before designing better hardware architectures. Operating system engineers need to understand how user-level programs

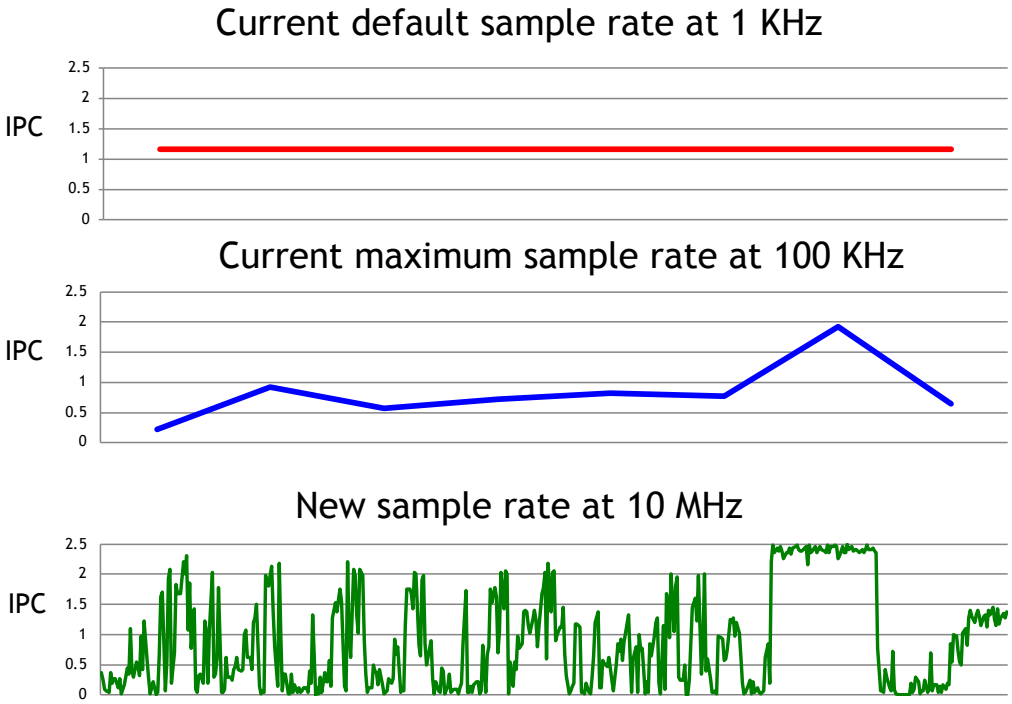


Figure 1.2: IPC timeline for Lusearch. Sampling with 10 MHz exposes behavior unseen by existing profilers (red, blue).

invoke kernel system calls before optimizing hot paths. Latency-critical application developers, from virtual reality (VR) to web services, have to diagnose root causes of tail latency before designing new optimizations.

General purpose profilers are the most popular and easy-to-use tools. They observe system behavior by periodically sampling software and hardware events. Unfortunately, due to low sample rates, popular state-of-the-art profilers, such as Linux Perf [Linux, 2014a] and Intel Vtune [Intel, 2014], are only able to observe corse-grain behavior. They are blind to the fine-grain behavior needed to explain many performance problems. Despite the fact that modern computer systems are running on gigahertz processors, the sample rates of current profilers have been fixed between 1 KHz and 100 KHz because of their disruptive sampling approach: they take an interrupt and sample hardware performance counters and software states. The possibility of overwhelming the operating system's capacity to service interrupts places limits on such profilers' maximum resolution. Low-frequency sampling misses both invaluable hardware events and software events. Figure 1.2 shows timelines of instructions per second (IPC), one important metric that reflects how software utilizes CPU micro-architectural resources, at different sample rates. Profilers sampling at 1 KHz and 100 KHz are not capable of showing fine-grain IPC variations.

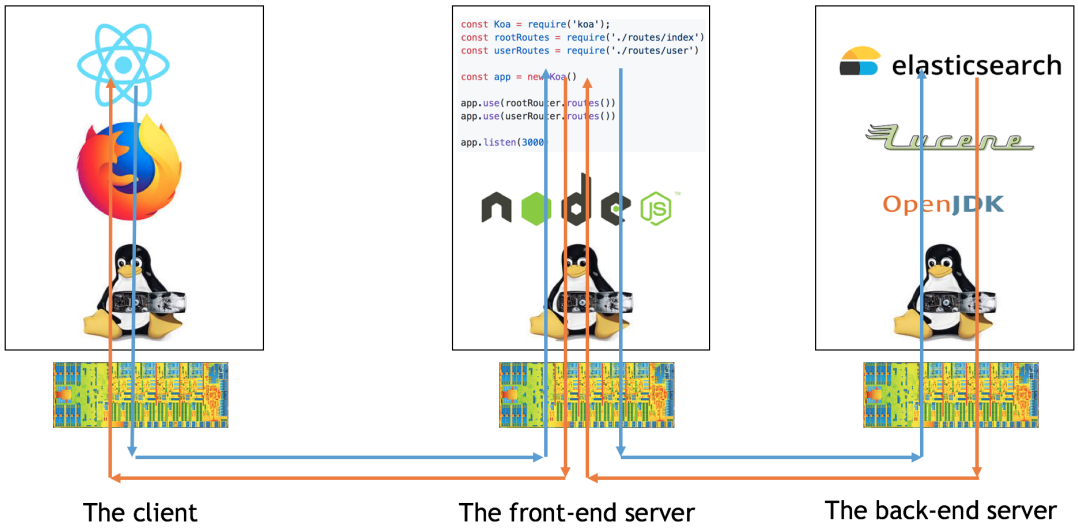


Figure 1.3: A simple web request crosses layers of complex systems in a few milliseconds.

Figure 1.3 shows that in a few milliseconds, a web request passes through multiple system stacks across three machines. Current general-purpose profilers are not able to analyze fine-grain interactions between system components when requests travel through stacks. To observe fine-grain system behavior, engineers reluctantly switch to other inconvenient and high-overhead approaches, such as microbenchmarks, simulation, direct measurement, tracing and exhaustively sampling. Because of such limitations, these alternative approaches cannot directly observe the behavior of online production systems.

1.2 Problem Statement

Understanding fine-grain system behavior is important for diagnosing performance problems and designing new optimizations. However, current general purpose profilers are not capable of observing the invaluable fine-grain system behaviors of online production systems, which slows down performance improvements.

1.3 Insights and Contributions

This thesis presents three contributions: 1) To observe the fine-grain hardware and software behavior of online production systems, we introduce Shim, a new continuous profiling approach that samples at resolutions as fine as 15 cycles, presents high-fidelity system signals, and automatically filters out noisy samples. 2) To analyze and control fine-grain interactions of complex system components in real time, we design Tailor, a Shim-based real-time latency controller. Tailor continuously samples signals of complex system components to identify system behaviors that can delay latency-critical requests and to take real-time control actions to reduce

the impact of these hazardous behaviors. 3) SHIM's real-time fine-grain control of system components enables a new class of feedback-directed optimizations. Many latency-critical workloads have low CPU utilization because they must consistently deliver low responses in order to retain users. We introduce ELFEN, a SHIM-based job scheduler that borrows the cycles in the idle periods of latency-critical workloads for batch workloads. ELFEN improves CPU utilization significantly without interfering with latency-critical requests by monitoring status changes of latency-critical requests with SHIM, and taking real-time scheduling actions. We next describe each of these contributions in more detail.

Shim The sample rates of current general purpose profilers are limited by their inherently disruptive sampling approach: they fire an interrupt and sample events in the interrupt handler. The overheads of taking interrupts in the kernel and the possibility of overwhelming the kernel's capacity to service interrupts place limits on their maximum resolution [Linux, 2014b].

SHIM views computer systems as high-frequency signal generators, and observes fine-grain system behavior by sampling generated hardware and software signals in a co-running observer thread at high frequencies on existing multicore hardware. A SHIM observer thread executes simultaneously with the application thread it observes, but on a separate hardware context, exploiting unutilized hardware on a different core or on the same core with simultaneous multithreading (SMT). Instead of using interrupts or inserting instrumentation, which substantially perturb applications, SHIM efficiently samples hardware and software signals by simply reading hardware counters and memory locations. SHIM improves its accuracy by automatically detecting and discarding samples affected by measurement skew which it or other system parts introduce. We measure SHIM's observer effects and show how to analyze them. When on a separate core, SHIM can continuously observe one software signal with a 2% overhead at a ~1200 cycle resolution. At an overhead of 61%, SHIM samples one software signal on the same core with SMT at a ~15 cycle resolution. We vary prefetching and DVFS policies in case studies that show the diagnostic power of fine-grain IPC and memory bandwidth results. By repurposing existing hardware, we deliver a practical tool for fine-grain performance microscopy for developers and architects.

Tailor Faced with complex computer systems, it is a challenge for latency-critical web services to consistently deliver low responses, because complex system components can unexpectedly delay latency-critical requests by orders of magnitude larger than their target latency. SHIM's high-frequency profiling enables a new approach to identifying and addressing these hazardous system behaviors in real time.

TAILOR is a SHIM-based real-time latency controller that continuously monitors and acts on hazardous system behaviors. TAILOR identifies such behaviors by recording and analyzing fine-grain interactions of system components. For unavoidable hazardous system behaviors, TAILOR uses local-server redundancy to mitigate their impact. TAILOR consists of two parts, a SHIM-based high-frequency profiler and an

application-level network proxy. The profiler continuously records request-related events from system components that show how the system stack processes latency-critical requests into a timeline stream. It also detects and takes real-time control actions on hazardous behaviors. The proxy forwards requests and responses for the active and backup servers. Whenever it detects a slow request, it searches and analyzes that request's related events from the timeline stream, then presents a timeline to developers showing how system behaviors affect the slow request. We evaluate TAILOR with a latency-critical Lucene workload whose client-side tail latency is 16 times larger than the maximum server time. We show that TAILOR identifies hazardous system behaviors that are from the operating system and the JVM. By adjusting system configurations to avoid the hazardous behaviors induced by the operating system and mitigating the impact of unavoidable JVM pauses with local-node redundancy, TAILOR reduces tail latency nine-fold, from 46 ms to 5 ms.

Elfen Not only does profiling critical workloads with SHIM show promising optimization opportunities, but SHIM's real-time fine-grain control of system components can help us implement a new class of real-time feedback-directed optimizations. For instance, profiling Lucene, a popular open-source search engine, shows that interactive web services have short idle periods and poor temporal locality between requests. This profiling result presents a promising optimization opportunity: we can improve datacenter utilization significantly without interfering with latency-critical requests if we can fill many short idle periods with batch jobs. However, the challenge of implementing the optimization is how we can accurately control batch jobs so that they enter and leave the short idle periods in real time. SHIM's high-frequency profiling and fine-grain control make it possible to detect short idle periods and take real time scheduling actions.

ELFEN is a SHIM-based scheduler that can co-run batch jobs and latency-critical requests on the same core but on different SMT contexts without interfering with latency-critical requests. ELFEN employs principled borrowing, a scheduling technique that dynamically identifies idle cycles and runs batch workloads by borrowing hardware resources from latency-critical workloads without violating SLOs. We instrument batch threads with profiling instructions that continuously monitor paired request lanes just as SHIM samples system signals. Batch threads start to execute only when their paired request lane is idle, quickly stepping out of the way when a latency-critical request starts executing. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when a request lane is idle improves utilization between 90% and 25% on one core, depending on load, without compromising request service level objectives (SLOs). Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

1.4 Thesis Structure

Designing optimizations is an experimental science. It follows three steps: observing computer systems, developing insights, and designing new optimizations. The body of this thesis is structured around this optimization discovery process.

Chapter 2 gives background and an overview of profilers. Chapter 3 explains SHIM, a new high-frequency sampling approach that can observe fine-grain system behavior. Chapter 4 describes TAILOR, a SHIM-based real-time latency controller that can analyze and control the fine-grain interactions of complex system components. Chapter 5 discusses ELFEN, a SHIM-based job scheduler that controls fine-grain interactions between latency-critical requests and batch jobs.

From more accurate microscopes that resulted in discovering bacteria to remarkably clear X-ray diffraction images of DNA that lead to identifying the DNA structure, the history of science shows that improvements in critical measurement techniques stimulate fundamental new discoveries. This thesis demonstrates that significant improvements in sampling rates also lead to new insights and optimizations.

Chapter 2

Background

Profilers help engineers observe the run-time behavior of systems, understand performance problems, and design new optimizations. They record time-varying software and hardware events, analyze these signals, and present profiling results. This chapter gives an overview of profilers, structured as follows: Section 2.1 explains how profiling works; Section 2.2 introduces the development of profiling techniques; Section 2.3 discusses the problem domain addressed by this thesis.

2.1 Profiling

Profilers observe the behavior of systems by recording and analyzing time-varying software and hardware events. This section first explains what and where these events are, then introduces how profilers read and analyze the events.

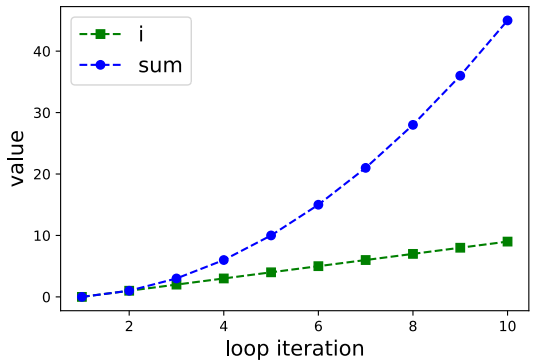
2.1.1 Software and Hardware Events

Computer systems continuously change software and hardware states. For example, when executing the code in Figure 2.1(a), the software states of variables `i` and `sum`, stored either in registers or memory, are updated at every loop iteration; hardware states, such as the number of retired instructions, are updated at every CPU cycle. If we view computer systems as signal generators, and places holding software and hardware states as signal channels, whenever computer systems (signal generators) mutate states, they generate events in signal channels. These time-varying software events and hardware events are signals of system behavior, as shown in Figure 2.1(b) and Figure 2.1(c). Thus, profilers are able to observe and analyze system behavior by reading events from signal channels and analyzing sampled signals.

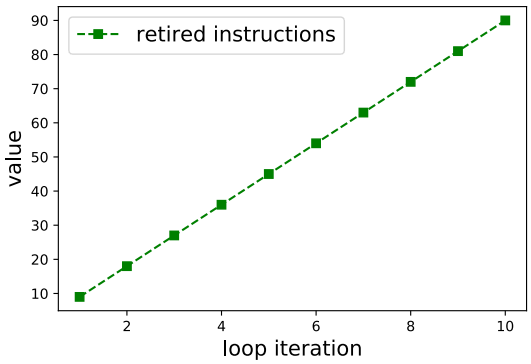
Software channels are storage locations holding the software state and interfaces for accessing the state, such as registers, memory locations, files and APIs. Some software channels are explicit about where and how to sample them. For example, Linux maintains each process's status and exports it via the `/proc/[pid]/stat` virtual file. But there are many program-specific software channels whose location is not explicitly exported, such as storage locations of variables `i` and `sum`, the loop and method identifiers in Figure 2.1(b). To sample program-specific channels, profilers

```
1 for (int i=0; i<10; i++)
2   sum += i;
```

(a) A simple loop



(b) Two Software signals



(c) A hardware signal

Figure 2.1: Computer systems generate software and hardware signals.

need to locate them first. Figure 2.2 shows the process of locating channels of the current-running-method event in a Java virtual machine on Linux. Profilers look up running Java threads from channels of currently running tasks, then search for their current frame pointer from channels of current frames, finally they read the method IDs from channels in the stack.

Hardware channels are places holding hardware states. On modern processors, the main software-accessible channels are performance counters and hardware profiling buffers. Modern processors provide a few configurable performance counters for each hardware thread (hardware context) to count architectural events. Each counter has a control register with which software chooses the event to be monitored from a rich set of architectural events such as the number of retired instructions, L1 data/instruction cache misses, and memory references. The control register also has flags to control whether to generate an interrupt on counter overflow, to select hardware domains (counting events from the current hardware thread or from the whole core), and other conditions. Once a counter is started, the processor will increase its value for matched events.

Figure 2.3 shows an Intel performance counter and its control register. On modern Intel processors, software chooses one event to count from hundreds of architectural events, and reads the counter value with the `RDMSR` instruction. In addition to counters, modern processors also provide hardware profiling buffers. For example, Intel PEBS [Intel, 2019] and AMD IBS [AMD, 2019] store sampled instructions and associated architectural performance states such as memory-reference latency in profiling buffers. Intel Processor Trace (Intel PT) [Strong, 2014] records full traces of instructions to trace buffers.

It is tedious to directly manage these hardware channels, thus operating systems and other libraries provide more friendly interfaces. For example, Linux provides the `sys_perf_event_open()` system call to manage performance counters and profiling

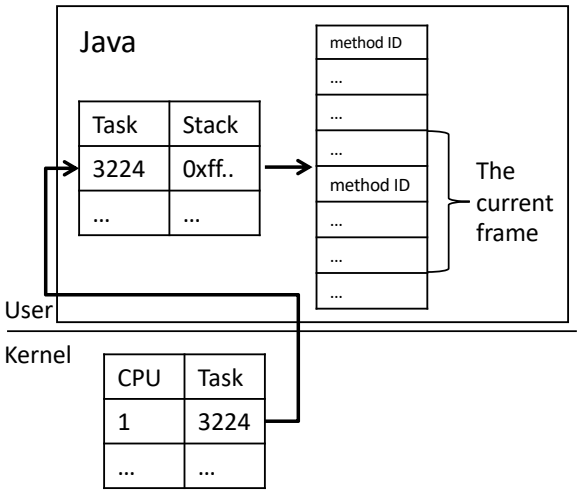
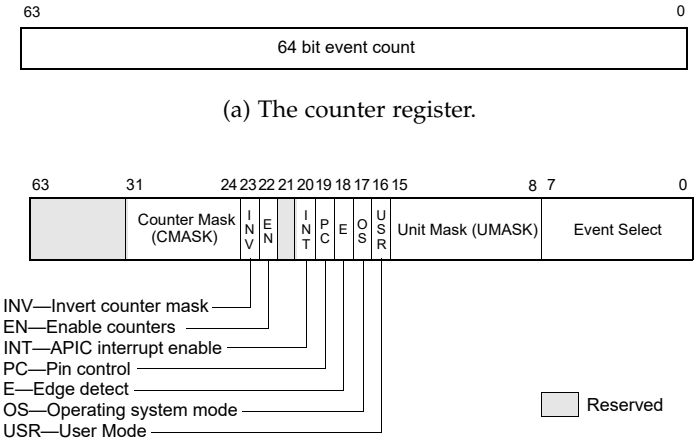


Figure 2.2: Locating signal channels of current-method-id events.



(a) The counter register.

(b) The control register.

Figure 2.3: The Intel performance counter specification [Intel, 2019].

buffers. Libpfm provides library interfaces to translate human-readable event names to hardware configurations.

2.1.2 Profiling Approaches

The quality of recorded signals determines the level of detail profilers can observe and analyze. Current profilers mainly use two approaches to record system signals: 1) Instrumentation: profilers instrument target programs with code that reads and analyzes hardware and software events. This approach is good at locating program-specific software channels and recording full traces of target events, and thus is adopted by many software profilers. 2) Interrupt-driven sampling: profilers send interrupts and execute code that reads and analyzes events in the interrupt handler. Because this approach can continuously sample the whole system at low frequencies but with low overhead and correlate sampled coarse-grain signals, it is the default profiling approach of current continuous system profilers.

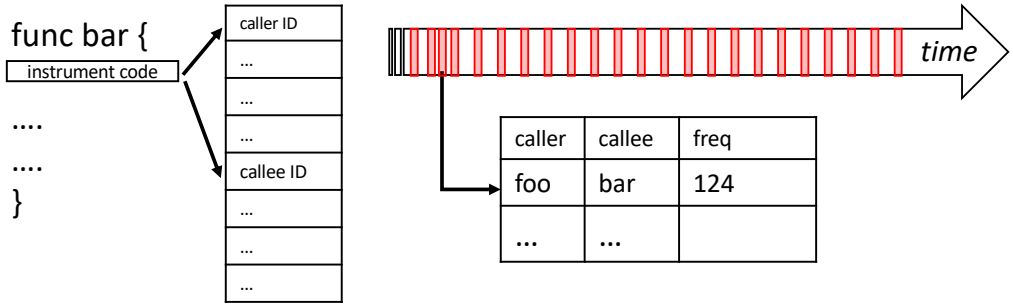
Modern computer systems generate high-frequency signals. However, because both of the above obtrusive approaches are not capable of recording high-frequency signals, current profilers are not able to observe and analyze fine-grain system behavior. This thesis introduces direct observation, a new non-obtrusive high-frequency sampling approach that continuously polls remotely observable events of target contexts in a separate hardware context. This new approach enables Shim, Tailor, and Elfen to observe, analyze, and control the fine-grain behavior of production systems.

Instrumentation Instrumentation code is added to programs to record hardware and software events at run time. The code can be added at any stage: engineers directly implement the code in the source code; compilers insert the code either in offline ahead-of-time (AOT) compilation or online just-in-time (JIT) compilation; binary modification tools insert the code to the final binary executable files.

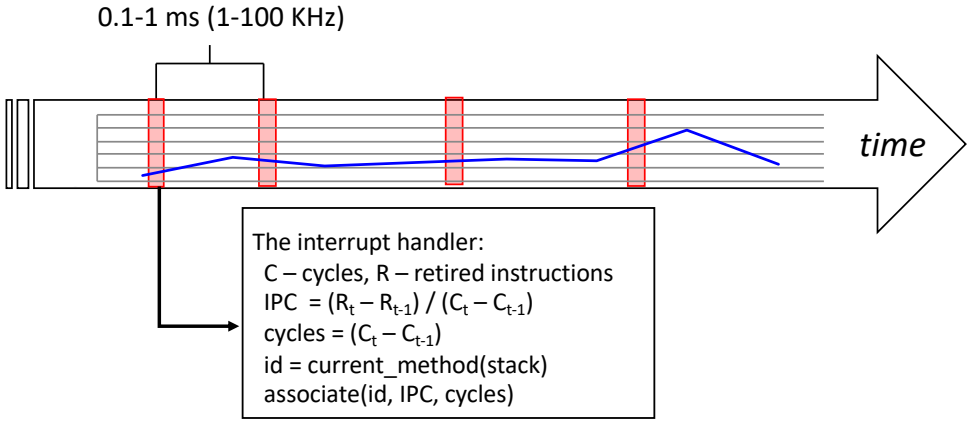
Figure 2.4(a) shows an example of how the instrumentation approach collects the call graph—important profiling statistics showing frequencies of invoking methods. The instrumentation code, inserted to the function prologue, reads the current method ID and its caller method ID from signal channels in the stack and updates the corresponding entry in the call graph.

The insertion of instrumentation code is normally done by tools that understand target programs, thus this approach is good at locating program-specific channels. As shown in the above example, since the compiler controls the stack layout, the instrumentation code inserted by the compiler knows where to read the caller and callee IDs. The instrumentation code is always invoked by the modified code, so it is capable of recording full traces of modified-code-related events. However, the instrumentation code is also only invoked by the modified code, which is likely to produce biased samples due to poor sample-space coverage.

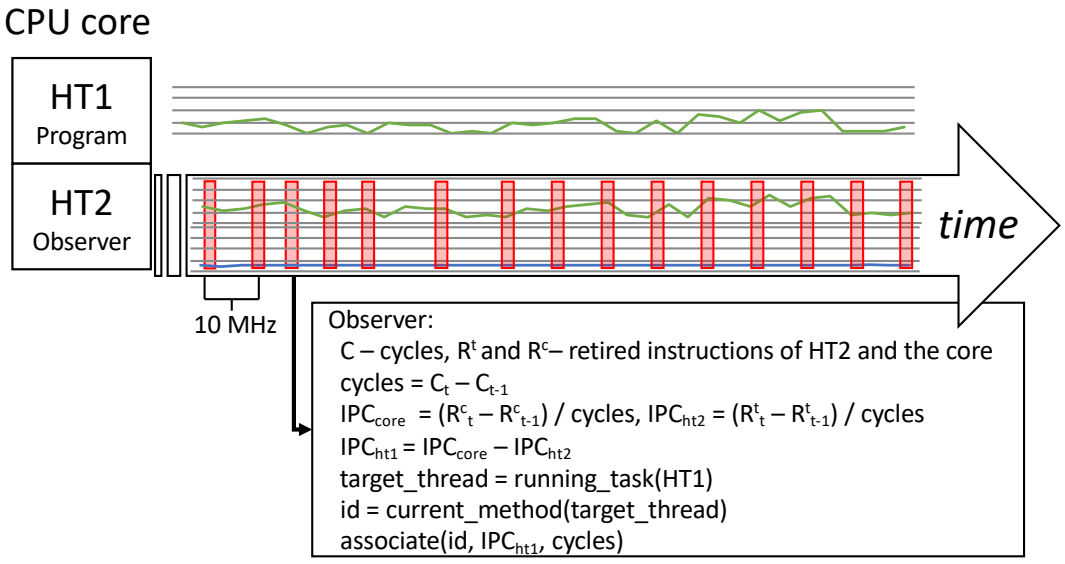
Instrumentation contributes direct overhead to target contexts. The overhead is determined by the performance and execution frequencies of instrumentation code.



(a) The instrumentation code reads the caller and the callee (software events) from the stack (signal channel) to compute the call graph.



(b) The interrupt handler samples and correlates IPCs, current method-ids, and cycles.



(c) The observer samples and correlates the target hardware thread's IPCs, method IDs, and cycles.

Figure 2.4: Three sampling approaches: instrumentation, interrupt-driven sampling, and direct observation.

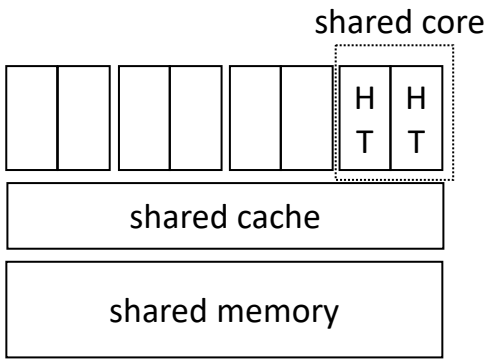


Figure 2.5: Shared hardware resources can be shared channels.

Therefore recording high-frequency system signals with this approach not only incurs unacceptable overhead, but also introduces significant observer effects.

Interrupt-driven Sampling The interrupt-driven sampling approach sends interrupts to target hardware contexts, and executes profiling code in the interrupt handler. Figure 2.4(b) shows how this approach associates hardware events of IPCs and cycles with software events of the current method. Profilers send periodic interrupts to the target CPU at low frequencies, between 1 KHz and 100 KHz. In the interrupt handler, the profiling code computes the IPC and cycles of the last sampling period, then associates them with the current method which is read from the current thread's stack.

Interrupt-driven sampling is the default profiling approach for current continuous system profilers because: 1) it avoids inserting instrumentation code; 2) it can cover a large set of system components by randomly adjusting sampling intervals over a long time, and 3) it is able to control its sampling overhead by limiting sampling frequencies. However, the possibility of overwhelming the operating system's capacity to service interrupts plus the overhead of taking interrupts places limits on sampling frequencies. As a result, this approach cannot observe fine-grain behaviors and accurately analyze sampled signals. As shown in Figure 2.4(b), because millions of cycles and thousands of methods can be executed in one sampling period, the average IPC of the sampling period neither shows high-frequency signals, nor accurately represents the IPCs of associated methods.

Direct Observation Both the instrumentation approach and the interrupt-driven approach execute the profiling code in target contexts, either by inserting the instrumentation code to target software or sending interrupts to target hardware threads to execute the instrumentation code. The overhead of executing injected profiling code prevents these approaches from observing fine-grain system behavior.

This thesis introduces direct observation, a non-obtrusive high-frequency sampling approach, it uses observer threads to sample events generated by target hard-

ware threads via channels hosted by shared resources on multicore processors as shown in Figure 2.5. The observer thread samples software events of other threads from remotely accessible software channels such as shared memory and shared files. For hardware events, it configures its performance counters to count hardware events of shared domains (e.g., a shared last level cache) and also can use hardware profiler buffers to record hardware events from other hardware threads.

Figure 2.4(c) shows how the observer thread samples similar events to the example in Figure 2.4(b). Two hardware threads, HT1 and HT2, share one simultaneous multithreading (SMT) core. The observer thread running on HT2 remotely samples events from HT1. It uses performance counters in its hardware context to compute the IPC and cycles of HT1, then associates them with the current method ID of the task running on HT1 which it reads from the task's stack in shared memory.

The direct observation approach samples signal channels at extremely high frequencies in a tight loop, and doing so does not introduce extra direct overhead to target contexts because it is non-obtrusive, which makes this approach a good candidate for high-frequency full-stack continuous profiling.

2.2 Profiling Techniques

General-purpose profilers were developed at the same time as early high-level languages: ALGO, FORTRAN, and C. These high-level languages encouraged engineers composing large complex programs with high-level abstractions such as FORTRAN statements, C functions and libraries. To attribute performance to activities with high levels of abstraction, compiler researchers and engineers started to build general-purpose profilers. Knuth [1971] uses The FORTRAN Debugging Aid Program (FORDAP) that can instrument source code of FORTRAN programs, to gather execution frequencies of FORTRAN statements over a set of selected FORTRAN programs. Satterthwaite [1972] implements an ALGOL W profiler by extending the ALGOL W compiler. The profiler inserts instrumentation code when compiling ALGOL programs to collect run-time statistics and traces of statements. Graham et al. [1982] designed gprof, a general-purpose profiler that records the run-time call graph and attributes execution time with C functions. They developed gprof in response to their efforts to improve their code generator [Graham, 1980]. Gprof adopts two profiling approaches: inserting instrumentation code in the function prologue to gather the call graph, and periodically sending timer interrupts to target programs to sample execution-time statistics of functions.

For modern managed high-level languages that support JIT compilation, their runtime normally has a built-in profiler for gathering program statistics which are fed into dynamic compilation for generating better code. Arnold and Grove [2005] design a low-overhead sampling-based profiler for Java virtual machines to gather an accurate call graph. Dynamic compilation and its supportive profiling can be implemented in hardware too. Transmeta's Cursoe [Dehnert et al., 2003], which dynamically translates x86 instructions to very-long-instruction-word (VLIW) code,

collects profiling statistics on execution frequency, branch directions, and memory-mapped I/O operations of source instructions.

Inserting instrumentation code into final binary executable files is more general and independent. MIPS's pixie, SUN's Spix, and qp/qpt [Larus and Ball, 1994] insert instrumentation code to binary executable files to gather run-time statistics and traces. ATOM [Srivastava and Eustace, 1994] and PIN [Luk et al., 2005] provide binary modification infrastructures, letting engineers customize their instrumentation code to collect different profling results.

To avoid reloading modified operating system images, Dtrace [Cantrill et al., 2004], Ftrace [Rostedt, 2019], and BCC [iovisor, 2019] reuse built-in instrumentation probes added by OS developers to dynamically load instrumentation code. They also provide script languages to help engineers develop customized instrumentation code.

Inserting instrumentation code is obtrusive and invoking the instrumentation code in hot paths introduces high overhead. Conte et al. [1996] design a low-overhead dedicated hardware buffer that is able to record statistics of conditional branches. Chen et al. [2006] proposes Log-Based Architectures (LBA) that is capable of recording and analyzing sofwate execution logs. commodity processors support hardware profiling buffers too. Intel PEBS [Intel, 2019] and AMD IBS [AMD, 2019] store sampled instructions and associated CPU states in profiling buffers. Intel Processor Trace [Strong, 2014] and ARM CoreInsight [ARM, 2019] record full traces of instructions.

Continuous profiling the whole stack, including hardware, the operating system, libraries and applications, can greatly help engineers understand system behavior, but it must have low overhead. DCPI [Anderson et al., 1997] and Morph [Zhang et al., 1997] are progenitors of today's interrupt-driven continuous profilers such as Vtune [Intel, 2014], Oprofile [OProfile, 2014], Linux Perf [Linux, 2014a], and Google GWP [Ren et al., 2010]. They periodically send interrupts and sample software and hardwaren events in the interrupt handler. To avoid overwhelming interrupt handling and control overhead, they samples at low frequencies, between 1 KHz and 100 KHz. Regular interval sampling produces biased samples [Mytkowicz et al., 2010; Bond and McKinley, 2007], DCPI introduced random sampling intervals to avoid the bias.

Modern computer systems have a complex system stack. Correlating events of components helps engineers find root causes of performance problems. Vertical Profiling [Hauswirth et al., 2004] correlates hardware, operating system, library, JVM, and application events. Ammons et al. [1997] attribute hardware events such as cache misses with hot paths. Kanev et al. [2015] continuously profile more than 20,000 Google machines over three years and correlate stack traces with hardware events.

2.3 Problem Domain

Modern computer systems are composed of layers of components, so they have a complex system stack even when the individual component seems simple. As shown in Figure 1.1, although a few lines of JavaScript can implement the HTTP server, the whole stack has millions of lines of code to support the application. Facing such

complex systems, identifying optimization opportunities is challenging. For example, Kanev et al. [2015] find that common components in the lower levels of the system stack can take 30% of cycles across Google servers after continuously profiling 20,000 servers over three years.

In this dissertation, we focus on building full-stack continuous profilers to help engineers continuously observe, analyze, and optimize production systems on existing hardware platforms, especially fine-grain complex interactions between system components.

2.4 Summary

This chapter gives an overview of profilers. We view time-varying software and hardware events as signals of system behavior. We compare three profiling approaches—instrumentation profiling, interrupt-driven sampling, and direct observation. We show the development of profiling techniques. Finally, we explain that this thesis focuses on designing profilers that enable engineers to observe, analyze, and control the fine-grain behavior of production systems. In the next three chapters, we demonstrate that Shim can continuously observe the hardware and software behavior of online production systems at a much finer granularity than prior work. We then show that this information is accurate and useful. We introduce Tailor, a Shim-based full-stack profiler that analyzes fine-grain complex interactions between system components, and Elfen, a Shim-based job scheduler, improves datacenter utilization by controlling fine-grain interactions between latency-critical requests and batch jobs.

Chapter 3

SHIM

Discovering performance optimizations is a challenging process: engineers observe system behavior, understand root causes, and then propose new optimizations. Unfortunately, the root causes of many problems occur at a finer granularity than existing profilers can observe. Limited by their interrupt-driven sampling approach, current profilers' sample rates are too low to observe the fine-grain system behavior of online production systems. In this chapter, we introduce SHIM, a new sampling approach that samples at resolutions three to five orders of magnitude finer than previous profilers. SHIM views computer systems as high-frequency signal generators, and observes fine-grain system behavior by sampling generated hardware and software signals at high frequencies in a profiling observer thread running on existing multicore hardware.

This chapter is structured as follows. Section 3.2 motivates fine-grain profiling by comparing coarse-grain and fine-grain sampling for hot methods, instructions per cycle (IPC), and IPC for hot methods. Section 3.3 introduces the design and implementation of SHIM. Section 3.4 examines SHIM's observer effects and shows how SHIM manages them to improve its accuracy. Section 3.5 describes our evaluation methodologies and Section 3.6 evaluates the strengths, limitations, and overheads of a variety of SHIM configurations and sampling rates.

The work described in this chapter is published in "Computer Performance Microscopy with SHIM" [Yang et al., 2015a].

3.1 Introduction

Understanding the complex interactions of software and hardware remains a daunting challenge. Developers currently use two main profiling approaches: instrumentation and interrupt-driven sampling. They pose hypotheses and configure these tools to instrument software events and read hardware performance counters. Next they attempt to understand and improve programs by correlating code with performance events, e.g., code to bandwidth consumption, low Instructions Per Cycle (IPC) to branch behavior, and loops to data cache misses. State-of-the-art continuous profiling tools, such as Intel VTune and Linux perf, take an interrupt and then sample hardware performance counter events [Intel, 2014; Linux, 2014a]. The possibility of

overwhelming the kernel's capacity to service interrupts places practical limits on their maximum resolution [Linux, 2014b]. Consequently, their default sample rate is 1 KHz and their maximum sample rate is 100 KHz, giving profile resolutions of around 30 K to 3 M cycles on a modern core.

Unfortunately, sampling at a period of 30 K cycles misses high frequency events. Statistical analysis sometimes mitigates this problem, for example, when a single small portion of the code dominates performance. However, even for simple rate-based measures such as IPC, infrequent samples are inadequate because they report the mean of the period, obscuring meaningful fine-grain variations such as those due to small but ubiquitous code fragments, as we show in Section 3.2.

Consequently, developers must currently resort to microbenchmarks, simulation, or direct measurement to examine these effects. However, microbenchmarks miss interactions with the application context. Simulation is costly and hard to make accurate with respect to real hardware. Industry builds proprietary hardware to examine performance events at fine granularities at great expense and thus such results are scarce.

The alternative approach directly measures code by adding instrumentation automatically or by hand [Linux, 2014a; Demme and Sethumadhavan, 2011; Zhao et al., 2008; Ha et al., 2009]. For instance, developers may insert instrumentation that directly reads hardware performance counters. Software profilers such as PiPA and CAB instrument code automatically to record events such as path profiles. A consuming profiler thread analyzes the buffer offline or online, sampling or reading it exhaustively. In principal, developers may perform direct measurement and fine-grain analysis with these tools [Ammons et al., 1997]. However, inserting code and the ~30 cycles it takes to read a single hardware performance counter both induce observer effects. Observer effects are inherent to code instrumentation and are a function of the number of measurements—the finer the granularity and the greater the coverage, the more observer effect. In summary, no current solution delivers accurate continuous profiling of hardware and software events with low observer effects at resolutions of 10s, 100s, or even 1000s of cycles.

This chapter introduces a new high resolution approach to performance analysis and implements it in a tool called Shim.[1] Shim efficiently observes events in a separate thread by exploiting unutilized hardware on a different core or on the same core using Simultaneous Multithreading (SMT) hardware. A Shim observer thread executes simultaneously with the application thread it observes, but in a separate hardware context.

Signals We view computer systems as high-frequency signal generators that generate high-frequency *signals*—time-varying software and hardware events—to *signal channels*, the places that hold hardware and software states. A Shim observer thread samples hardware signals from hardware channels such as performance counters, and

[1] A shim is a small piece of material that fills a space between two things to support, level, or adjust them.

software signals from software channels (e.g. memory locations that store method and loop identifiers). A compiler or other tool configures software signals and communicates memory locations to SHIM. SHIM treats software and hardware data uniformly by reading (sampling) memory locations (software channels) and performance counters (hardware channels) together at very high frequencies, e.g., 10s to 1000s of cycles. The observer thread logs and aggregates signals. Then further online or offline analysis acts on this data.

Measurement fidelity and observer effects The fidelity of continuous sampling is subject to at least three threats: (i) skew in measurement of rate metrics, (ii) observer effects, and (iii) low sample rates. SHIM reduces these effects.

To improve the fidelity of rate metrics, we introduce *double-time error correction* (DTE), which automatically identifies and discards noisy samples by taking redundant timing measurements. DTE separately measures the period between the start of two consecutive samples and the period between the end of the samples. If the periods differ, the measurement was perturbed and DTE discards it. By only using timing consistency, DTE correctly discards samples of rate measurements that seem obviously wrong (e.g., IPC values of > 10), without explicitly testing the sample itself.

A minimal SHIM configuration that only reads hardware performance counters or software signals inherent to the code does not instrument the application, so has no direct observer effect. SHIM induces secondary effects by contending for hardware resources with the application. This effect is largest when SHIM shares a single core with the application using Simultaneous Multithreading (SMT) and contends for instruction decoding resources, local caches, etc. We find that the SHIM observer thread offers a constant load, which does not obscure application behavior and makes it possible to reason about SHIM's effect on application measurements. When SHIM executes on a separate core, it interferes much less, but it still induces effects, such as cache coherence traffic when it reads a memory location in the application's local cache. This effect is a function of sampling and memory mutation rates. SHIM does not address observer effects due to invasive instrumentation, such as path profiles.

Randomization of sample periods is essential to avoiding bias [Anderson et al., 1997; Mytkowicz et al., 2010]. We show high frequency samples are subject to many perturbations and their sample periods vary widely.

We measure a variety of configurations and show that SHIM delivers continuous profiles with a rich level of detail at very high sample rates. On one hand, SHIM delivers ~15 cycle resolution profiling of one software signal in memory on the same core with SMT at a 61% overhead. Placing these results in context, popular Java profilers, which add instrumentation to applications, incur typical overheads from 10% to 200% at 100 Hz [Mytkowicz et al., 2010], with sample periods six orders of magnitude longer than SHIM. Because SHIM offers a constant load on SMT, SHIM observes application signals with reasonable accuracy despite its overhead. To fully validate SHIM's fine-grain accuracy would require ground truth from proprietary hardware-specific manufacturer tools, not available to us.

On a separate core, SHIM delivers ~1200 cycle resolution when profiling the same

software signal with just 2% overhead.

Case Studies The possibility of high fidelity, high frequency continuous profiling invites hardware innovations such as ultra low latency control of dynamic voltage and frequency scaling (DVFS) and prefetching to tune policies to fine-grained program phases. We analyze the ILP and bandwidth effects of DVFS and turning off and on prefetching on two examples of performance-sensitive code, showing that fine-grain policies have the potential to improve efficiency.

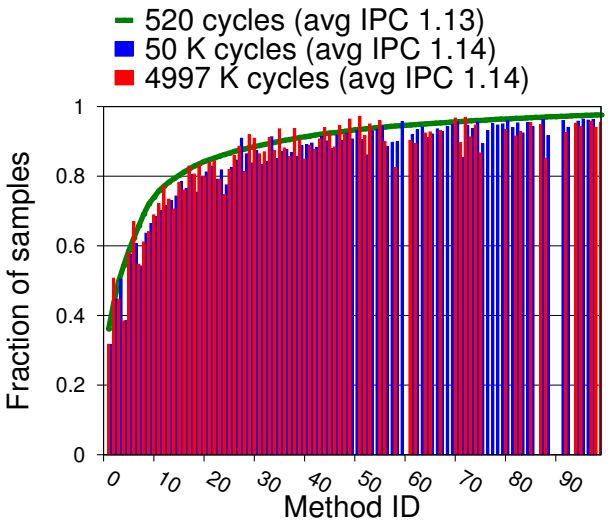
Hardware to improve accuracy and capabilities Modest hardware changes could significantly reduce Shim's observer effects, improve its capabilities, and make it a powerful tool for architects as well as developers. When Shim shares SMT execution resources, a thread priority mechanism, such as in MIPS and IBM's Power series [Snavely et al., 2002; Boneti et al., 2008], would reduce its observer effect. Executing Shim with low priority could limit the Shim thread to a single issue slot and only issue instructions from it when the application thread has no ready instruction. When Shim executes on a separate core, a no-caching read, such as on ARM hardware [Stevens, 2013], would reduce observer effects when sampling memory locations. A no caching read simply transfers the value without invoking the cache coherence protocol. On a heterogeneous architecture, the simplicity of the Shim observer thread is highly amenable to a small, low power core which would reduce its impact. If hardware were to expose all performance counters to other cores, such as in IBM's Blue Gene/Q systems [Bertran et al., 2013], Shim could capture all events while executing on a separate core. We show that Shim in this configuration would incur essentially no overhead and experience very few observer effects.

By repurposing existing hardware, Shim reports for the first time fine-grain continuous sampling of hardware and software events for performance microscopy. We make Shim publicly available [Yang et al., 2015b] to encourage this line of research and development.

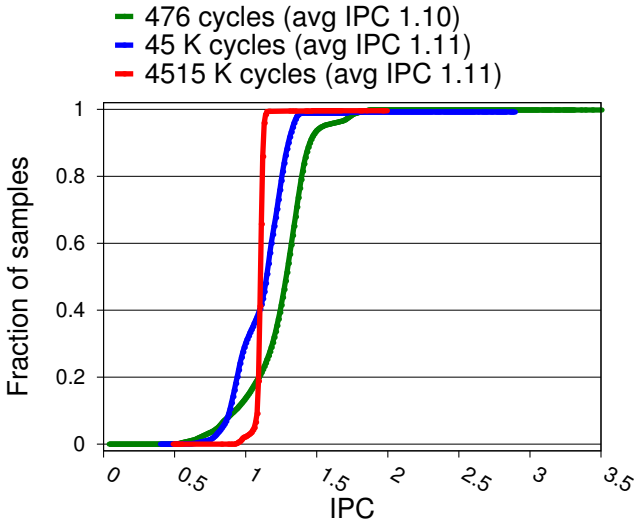
3.2 Motivation

This section motivates fine-grain profiling by comparing coarse-grain and fine-grain sampling for hot methods, instructions per cycle (IPC), and IPC for hot methods.

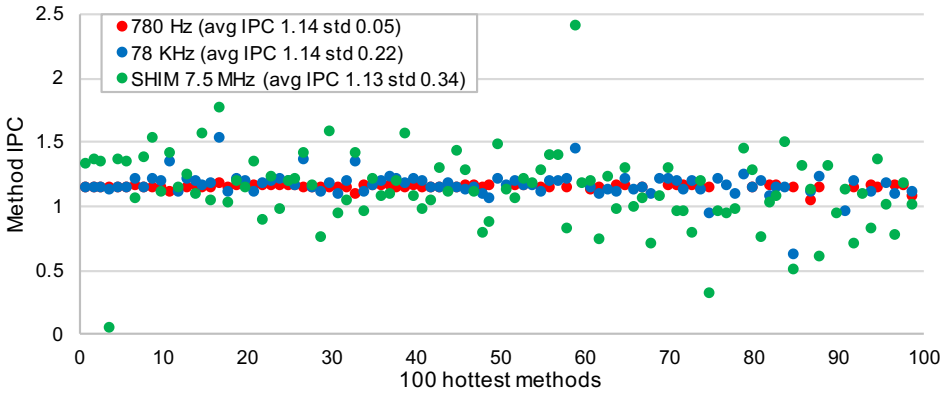
Identifying hotspots Figure 3.1(a) lists the 100 most frequently executed (hot) methods of the Java application lusearch for three sampling rates, ranging from ∼500 cycles to ∼50 K cycles to ∼5 M cycles. The medium and low frequency data are subsamples of the high frequency data. These results support the conventional wisdom that sample rate is not a significant limitation when identifying hot methods. The green curve is the cumulative frequency distribution of the number of samples taken for the hottest 100 methods when sampling at the highest frequency. The leftmost point of the green curve indicates that the most heavily sampled method accounts for



(a) Hot methods



(b) Instructions Per Cycle (IPC)



(c) IPC of hot methods

Figure 3.1: The impact of sample rate on lusearch. (a) Varying the sample rates identifies similar hot methods. The green curve is the cumulative frequency distribution of samples at high frequency. Medium (red) and low (blue) frequency sampling cumulative frequency histograms are permuted to align with the green. (b) Sample rate significantly affects measures such as IPC. (c) Sample rate significantly affects IPC of hot methods. Each bar shows average IPC for a given hot method at one of three sample rates.

36% of all samples, while the rightmost point of the curve reveals that the 100 most sampled methods account for 97% of all samples. The blue and red bars are the cumulative frequency histograms for medium and low frequency sampling respectively, reordered to align with the green high frequency curve. A gap appears in the blue and red histograms whenever a method in the green hottest 100 does not appear in their top 100 method histogram. The red histogram is noisy because many methods attract only two samples. However, the blue histogram is very well sampled, with the least sampled method attracting 142 samples. Most bars fall below the green line, indicating that they are relatively under sampled at lower frequencies, although a few are over sampled. While sample rate does not greatly affect which methods fall in the top 50 hottest methods, Mytkowicz et al. [2010] show that randomization of samples is important to accurately identifying hot methods.

Revealing high frequency behavior On the other hand, Figures 3.1(b) and 3.1(c) show that sampling at lower frequencies masks key information for metrics that vary at high frequencies, using IPC as the example. Figure 3.1(b) presents IPC for retired instructions. We measure cycles and instructions retired since the last sample to calculate IPC. The figure shows the cumulative frequency distribution for IPC over all samples. The green curve shows that when the sample rate is higher, SHIM observes a wider range of IPCs. About 10% of samples observe IPCs of less than 0.93 and 10% observe IPCs of over 1.45, with IPC values as low as 0.04 and as high as 3.5. By contrast, when the sample period grows, observed IPCs fall in a very narrow band between 1.0 and 1.3, with most at 1.11. As the sample period grows, each observation asymptotically approaches the mean for the program.

Figure 3.1(c) illustrates a second source of error due to coarse-grain sampling of rate-based metrics. In Figure 3.1(c), we calculate IPC, attribute it to the executing method, and then plot the average for each of the hottest 100 methods. Lowest frequency sampling (red) suggests that IPC is very uniform, at around 1.14, whereas high frequency sampling (green) shows large variations in average IPC among methods, from 0.04 (#4), to 2.39 (#59). The lower IPC variation at lower sample rates is largely due to the fact that IPC is measured over a period. As that period grows, the IPC reflects an average over an increasingly large part of the running program. This period is typically much larger than the method to which it is attributed. When the period is orders of magnitude longer than the method's execution, sampling loses an enormous amount of information, quantified by the standard deviations at the top of Figure 3.1(c). This problem occurs whenever a rate-based measure is determined by the period of the sample.

Direct measurement avoids this problem [Ammons et al., 1997; Pettersson, 2003; Linux, 2014a; Demme and Sethumadhavan, 2011], but requires instrumenting the begin and end of each method in this example. When just a few methods are instrumented, this method can work well, but when many methods are instrumented, methods are highly recursive, or methods execute only for a few hundred cycles, taking measurements (the observer effect) will dominate, obscuring the context in which the method executes.

3.3 Design and Implementation

Viewing time-varying events as signals motivates our design. A SHIM observer thread executes continuously in a separate hardware context, observing events from an application thread executing on neighbor hardware. The observer samples hardware and software signals from signal channels at extremely high frequencies, logging or analyzing samples depending on the configuration. SHIM samples signals that the hardware automatically generates in performance counters (hardware channels) and memory locations (software channels) that the application either explicitly or implicitly generates in software. SHIM consists of three subsystems: a coordinator, a sampling system, and a software signal generating system.

Signals SHIM observes signals from either hardware or software for three kinds of events: *tags*, *counters*, and *rates*. An event *tag* is an arbitrary value, such as a method identifier, program counter, or stack pointer. An event *counter* increments a value each time the event occurs. Hardware counters can choose events to be monitored from a rich set of performance events supported by the processor, including cycles, instructions retired, cache misses, prefetches, and branches taken. Software similarly may count some events, such as allocations, method invocations, and loop iterations. Software signals may be implicit in the code already (e.g., a method identifier or parameter on the stack) or a tool may explicitly add them. For example, the compiler may insert path profiling code. SHIM computes *rates* by reading a given counter X and the clock, C, at the start and end of a sample period and then computing the change in X over change in C for that period. Section 3.4 describes how SHIM correctly reports rates by detecting and eliminating noise in this process.

Coordinator The coordinator configures the hardware and software signals, sampling frequency, analysis, and the observer thread(s) and the location(s) on a different core on a Chip Multiprocessor (CMP) or the same core with Simultaneous Multithreading (SMT) as the application thread(s). The coordinator configures hardware performance counters by invoking the appropriate OS system calls. It invokes the software signal generation system to determine memory addresses of software signal channels. The coordinator communicates the type (counter or tag) of each signal, hardware performance counters, and memory locations for software signal channels to the observer thread. The coordinator binds each SHIM observer thread to a single hardware context. For each observer thread, we assign a *paired neighbor* hardware context for application thread(s). The coordinator executes one or more application threads on this paired neighbor. The coordinator starts the application and observer execution. We add to the OS a software signal channel that identifies application threads, such that SHIM can differentiate multiple threads executing on its paired neighbor, attributing each sample correctly. SHIM thus observes multithreaded applications that time-share cores.

Sampling system The SHIM observer thread implements the sampling system. It observes some number of hardware and software signals at a given sampling rate, as configured by the coordinator. The sampling system observes hardware signals by reading performance counters and the software signals by reading memory locations. The coordinator initializes the sampling system by creating a buffer for samples. The observer thread reads the values of the performance counters and software addresses in a busy loop and writes them in this buffer, as shown in Figure 3.2.

We divide the observer into two parts, one for counters (lines 4 to 9) and another for tags (lines 13 to 16). Software or hardware may generate counters, rates, or tag signals. Recording counters and rates requires high fidelity in-order measurements. We use the `rdtscp()` instruction, which returns the number of cycles since it has been reset. It forces a synchronization point, such that no read or write may issue out of order with it. It requires about 20 cycles to execute.

Each time SHIM takes one or more counter samples, it first stores the current clock (line 4 in Figure 3.2). It then synchronously reads every counter event from either hardware performance counters or a software specified memory location and then stores the clock again (line 9). We can measure rates by comparing values read at one period to those read at the previous one. The difference in the clock tells us the period precisely. If the time to read the counters (lines 4 to 9) varies from period to period, the validity of the rate calculation may be jeopardized. As we explain in Section 3.4, we can precisely quantify such variation and filter out affected measurements. Because it is correct to sample any tag value within the period, we do not read tags synchronously (lines 13 to 16).

The simple observer in Figure 3.2 stores samples in a buffer. Realistic observers will use bounded buffers which are periodically consumed or written to disk, or they may perform lightweight online analysis such as incrementing a histogram counter. Chapter 4 shows one example of realistic observers, TAILOR that correlates samples from different system components online. Real-time feedback directed optimizers can process and act on such data, Chapter 5 shows ELFEN, a SHIM-based scheduler, continuously monitors the status of latency-critical requests, and makes real-time scheduler decision based on the status.

Signal system The signal system generates software signals and selects hardware and software signals.

For hardware signals, we choose and configure hardware performance counter events. These configurations depend on the analysis and on which core the observer executes. For example, when observing the number of cycles the application thread executes, we only need the elapsed cycles event counter, when all hardware threads execute at the same frequency. To measure how many instructions the application executes, we need two counters on a two-way SMT processors to derive what happens on the neighbor thread. One counter counts instructions retired by the whole core and another one counts instructions retired by the SHIM thread. The difference is due to the application thread.

For software signals, we record the address of software channels where the appli-

```
1  void shimObserver() {
2    while(1) {
3      index = 0;
4      buf[index++] = rdtscp();    // counters start marker
5      foreach counter (counters){ // hardware or software counter
6        rdtscp(); //serializing instruction
7        buf[index++] = rdpmc(counter) or read_signal(counter);
8      }
9      buf[index++] = rdtscp();    // counters end marker
10     // which application thread is executing the paired neighbor?
11     pid_and_ttid = *pidsignal;
12     buf[index++] = pid_and_ttid;
13     if (tags){
14       foreach tag (tags)        // hardware or software tag
15         buf[index++] = rdpmc(tag) or read_signal(tag);
16     }
17     // online analysis here, if any
18   }
19 }
```

Figure 3.2: SHIM observer loop.

cation writes the software signal. Applications and runtimes already generate many interesting software signals automatically. For example, consider recording the memory allocation throughput to correlate it with cache misses. Many runtimes use bump pointers which are a software signal reflecting memory allocation. As we explain in Section 3.6.1, some JVMs also naturally produce a signal that reflects execution state at a 15-150 cycle resolution. Of course if the address of a software channel changes, it needs to be communicated to SHIM. Note that updating a memory address after say, every garbage collection, will be less intrusive than instrumenting every write of this address for frequently written variables.

For software signals that require instrumentation, we modify the Jikes RVM compiler. For each signal, the program simply writes the value in the same memory address (software channel) repeatedly. We experiment below with method and loop identifier signals and show that even though they occur frequently, they incur very low overheads on CMP. Because software signals write the same location, they exhibit good locality. Furthermore, most modern architectures use write-back polices, which coalesce and buffer writes, executing them in very few cycles.

Adding the instrumentation for software signals for highly mutating values in the SHIM framework incurs less observer effect than the instrumentation approach, which both mutates the value and then typically writes it to a distinct buffer location, rather than overwriting a single memory location.

This same software instrumentation mechanism may communicate hardware register state that is only architecturally visible to the application thread, e.g., the program counter and stack pointer could be written to memory as a software signal, but we leave that exploration to future work.

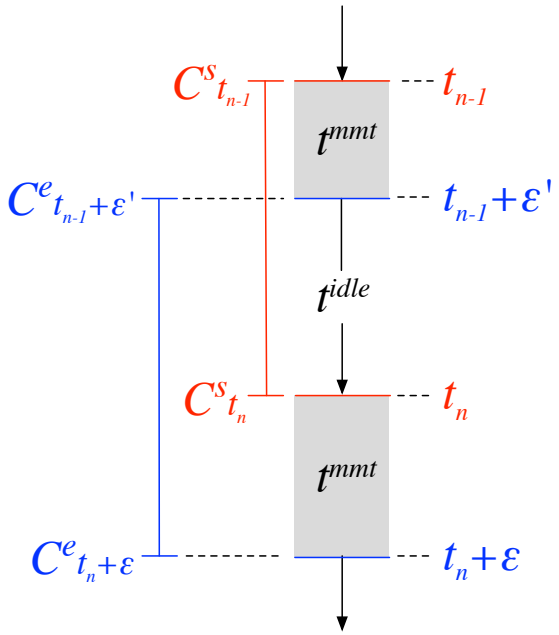


Figure 3.3: Four clock readings ensure fidelity of rate measurements. Grey regions depict two measurements, t_n^{mmt} and t_{n-1}^{mmt}, in which SHIM reads all counters. The sample period is, C^s (red) to C^e (blue). If the ratio of red and blue periods is one, then $t_n^{mmt} = t_{n-1}^{mmt}$ and SHIM does not induce noise. DTE discards noisy measurements of rate metrics based on this ratio.

3.4 Observation Fidelity

This section examines SHIM's observer effects and shows how SHIM manages them to improve its accuracy. This section describes and illustrates SHIM's (i) *double-time error correction* (DTE) of samples of rate metrics; (ii) sample period randomization; and (iii) some of its observer effects.

3.4.1 Sampling Correction for Rate Metrics

Many performance events are *rate-based*. For example, IPC relates retired instructions and clock ticks with respect to a time interval. The two major considerations for rate metrics are: (1) attributing a rate to one tag in the interval from possibly many tags for discrete semantic events, and (2) ensuring fidelity of the measure in the face of noise and timing skew.

Attribution of Tags to Sample Periods Although tags, such as method identifiers, occur at discrete moments in time, counting metrics are calculated with respect to a *period* (e.g., $C_{t_n} - C_{t_{n-1}}$ in Figure 3.3 explained below). SHIM reads each tag signal once during a sample period, and then attributes it to counters in that same period. Tags correspond to the correct period, but not to any particular point within that period. SHIM reads all hardware and software tags immediately after it has completed reading each counter value (i.e., after $t_{n-1} + \epsilon'$).

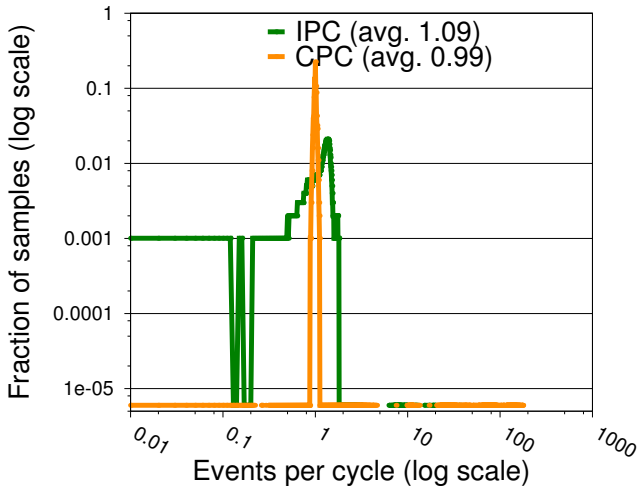
Rate Metrics Rates depend on *four* measurements. For example, IPC requires two instructions retired (IR) and two clock (C) measurements: $IR_{t_n} - IR_{t_{n-1}} / C_{t_n} - C_{t_{n-1}}$, ostensibly at times t_n and t_{n-1}. Since hardware can not read both simultaneously, SHIM takes each measurement at a slightly different time, t_n and $t_n + \epsilon$, resulting in: $IR_{t_n+\epsilon} - IR_{t_{n-1}+\epsilon'} / C_{t_n} - C_{t_{n-1}}$, at times t_n, $t_n + \epsilon$, t_{n-1}, and $t_{n-1} + \epsilon'$. Figure 3.3 shows two overlapping intervals in red and blue. For accurate rate-based measurements, we must bound the skew between the intervals $[t_n, t_{n-1}]$ and $[t_n + \epsilon, t_{n-1} + \epsilon']$.

The time to take the measurements, t^{mmt} plus the time spent idle, t^{idle} defines the sample period. Variance in t^{idle} is not problematic, in fact, it helps remove bias. The intervals $[t_n, t_{n-1}]$ and $[t_n + \epsilon, t_{n-1} + \epsilon']$ both cover a *single* t^{idle}, so variation in t^{idle} cannot introduce skew between $[t_n, t_{n-1}]$ and $[t_n + \epsilon, t_{n-1} + \epsilon']$. On the other hand, the intervals $[t_n, t_{n-1}]$ and $[t_n + \epsilon, t_{n-1} + \epsilon']$ encompass *two* measurement periods, t_n^{mmt} and t_{n-1}^{mmt} of the exact same measurements, so variation in either measurement period introduces skew. When the sample rate is high t^{idle} becomes small, and variation in t^{mmt} may dominate. Variation in t^{mmt} will thus be exposed as t^{idle} approaches t^{mmt} and can introduce skew, which as we show next undermines the fidelity of rate-based measures.

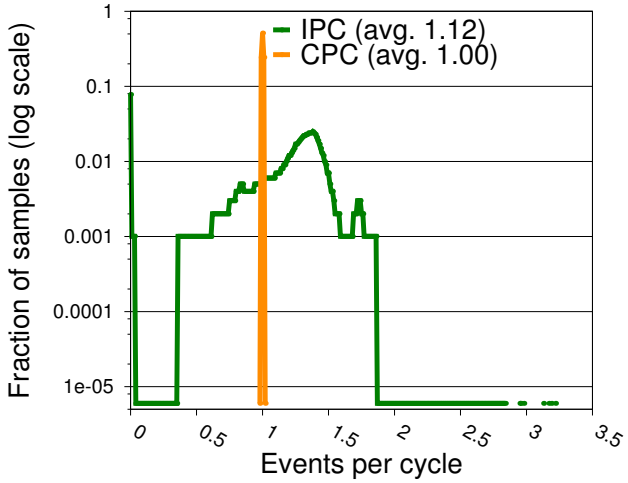
DTE Filtering of Rate Metrics We introduce *double-time error correction* (DTE) to correct skew in rate metrics. DTE takes the *two* clock measures, C^s and C^e for each measurement period t^m, one at the start, and one at the end (lines 4 and 9 of Figure 3.2). The rate-based measure $C^e_{t_n+\epsilon} - C^e_{t_{n-1}+\epsilon'} / C^s_{t_n} - C^s_{t_{n-1}}$ precisely identifies measurement skew.

Note that CPC=$C^e_{t_n+\epsilon} - C^e_{t_{n-1}+\epsilon'} / C^s_{t_n} - C^s_{t_{n-1}}$ will be 1.0 when the two clock readings are not distorted. Since they measure the same idle period, if CPC=1 the time to take the two distinct measurements t^{mmt} is the same. DTE uses this measure to identify statistically significant variation in t^{mmt} and discards affected samples. DTE therefore automatically discards noisy samples with significant variations in ϵ and ϵ' since they cannot correctly compute rate metrics. Figures 3.4(a) and (b) show the effect of DTE. (Section 3.5 describes methodology.) The graphs plot on log scales IPC in green and cycles-per-cycles (CPC) in orange. CPC=1 is ground truth. Figure 3.4(a) shows that before filtering values are clearly distorted—CPC is as high as 175 and IPC is 37 on a 4-way superscalar processor. Figure 3.4(b) shows IPC samples after DTE discards all samples with CPC values outside a $\pm$1% error margin. DTE filtering transforms CPC to an impulse at 1.0 (by design) and eliminates all IPC values greater than 4 (which we know are wrong without explicitly testing). All these wrong rates were introduced by sampling skew, which DTE detects and eliminates.

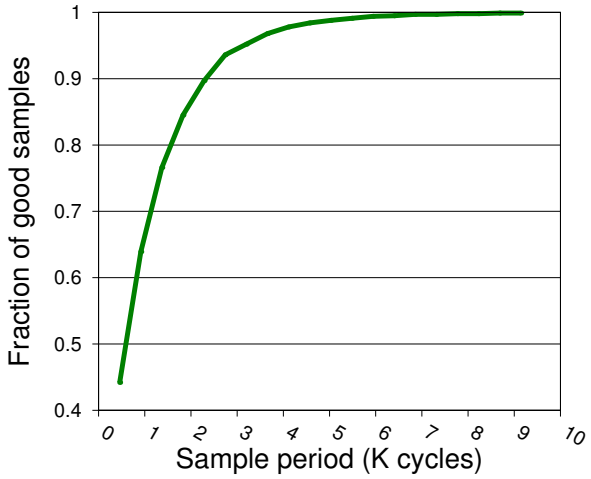
Figure 3.4(c) shows the fraction of good samples with DTE. At the highest frequency, DTE discards over 50% of samples, but at periods of $\geq$2500, 90% or more of samples are valid.



(a) Unfiltered IPC and CPC distortions, e.g., IPC> 4 on a 4-way superscalar processor.



(b) DTE correction with ±1% CPC error eliminates all distorted IPC values.



(c) Fraction of good samples with ±1% CPC error as a function of sampling frequency.

Figure 3.4: DTE filtering on SMT keeps samples for which ground truth CPC is 1.±0.01, eliminating impossible IPC values. At small sample periods, DTE discards over half the samples. At sample periods >2000, DTE discards 10% or fewer samples.

3.4.2 Randomizing Sample Periods

Prior work shows regular sample intervals in performance profiling, compared to random intervals, introduces observation bias [Anderson et al., 1997; Mytkowicz et al., 2010]. Figure 3.5 plots Shim's variation in sample period and gap on a log/log scale for SMT. We plot lusearch, but the results are similar for the other DaCapo benchmarks. The figure plots the frequency distribution histogram of sample periods ($C^s_{t_n} - C^s_{t_{n-1}}$) in green and the gap between samples in red. Figure 3.5 shows that there is enormous variation in sample period and the gap. The most common sample period, ~500 cycles, reflects only one percent of samples, and the gap between samples ranges from ~350 to ~49,000 cycles. Both SMT and CMP show a wide degree of variation, although different. This result gives us confidence that the hardware is naturally inducing large amounts of randomness in Shim's sampling, and thus Shim avoids the sampling bias problem due to regular intervals.

3.4.3 Other Observer Effects

The sampling thread has observer effects because it executes instructions, competing and sharing hardware resources with the application. Many of Shim's effects depend on how many and often Shim samples memory locations and counters, and the hardware configuration (SMT or CMP). Section 5.5 systematically quantifies many of these effects.

This section examines the behavior of the sampling thread itself to determine whether we can reason about it more directly. Figure 3.6 plots the SMT effect of Shim on IPC, on a log scale. The blue curve shows IPC for the whole core while the red curve shows the IPC just for Shim. The IPC of the Shim thread is extremely stable. The underlying data reveals that when operating with a 476 cycle period, 67% of all Shim samples are attributed to IPCs of 0.48 to 0.51 and 99% to IPCs of 0.54 to 0.43. By contrast, the IPC of the workload is broadly spread. The uniformity of Shim's time-varying effect on the core's resources makes it easier to factor out Shim's contribution to whole core measurements by simply subtracting its private counter value. For each hardware metric, developers can plot Shim, the total, and the difference to reason about observer effects.

Summary Shim corrects skewed samples of rates, randomizes sample periods, and offers a constant instruction execution observer effect on SMT. These features reduce, especially when compared to interrupt-driven and instrumentation profiling, but do not eliminate, observer effects.

3.5 Methodology

The evaluation in this chapter uses the following methodologies.

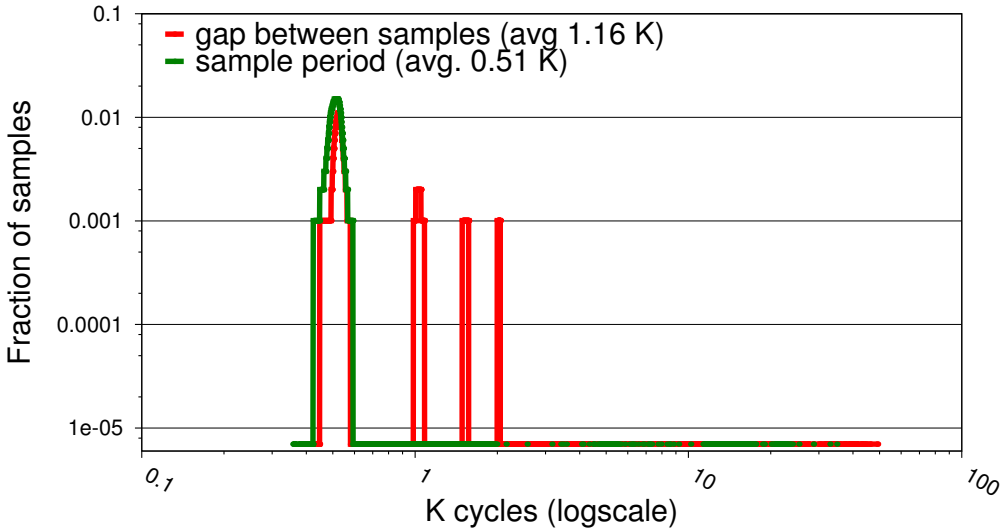


Figure 3.5: SHIM has large variation in sample period and between samples with DTE filtering. The green curve shows variation in the period of good samples. The red curve shows variation in the period between consecutive good samples.

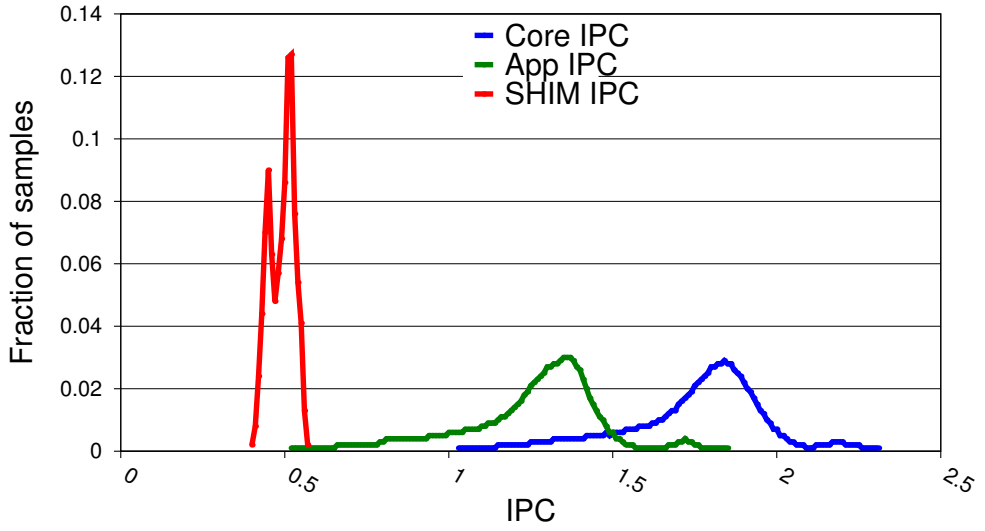


Figure 3.6: SHIM SMT observer effect on IPC for 476 cycle sample period with lusearch. The green curve shows lusearch IPC, red shows SHIM's IPC, and blue shows IPC for the whole core.

Software implementation We implement Shim in Jikes RVM [Alpern et al., 2005], release 3.1.3 + hg r10718, a Java-in-Java high performance Virtual Machine. We implement all of the functionality described in Section 5.3 by adding coordinator functionality, by modifying the VM scheduler, and by inserting signals with the compiler and VM. All measurements follow Blackburn et al.'s best practices for Java performance analysis [Blackburn et al., 2006a]. We use the default generational Immix collector [Blackburn and McKinley, 2008] with a heap size of six times the minimum for each benchmark and a 32 MB fixed size nursery to limit full heap collections to focus on application code in this chapter. We measure an optimized version of the code using *replay compilation*. Jikes RVM does not have an interpreter: it uses a baseline compiler to JIT code upon first execution and then recompiles at higher levels of optimization when a cost model predicts the optimized code will amortize compilation cost in the future [Arnold et al., 2000]. We record the best performing optimization plan, replay it, execute the resulting code once to warm up caches, then we iterate and only report measurements from this third iteration. We run each experiment 20 times and report the 95% confidence interval.

Benchmarks We draw benchmarks from DaCapo [Blackburn et al., 2006a], SPECjvm98 [SPEC, 1999], and `pjbb2005` [Blackburn et al., 2006b] (a fixed workload version of SPECjbb2005 [SPEC, 2006] with 8 warehouses and 10,000 transactions per warehouse.) The DaCapo and `pjbb2005` benchmarks are non-trivial real-world open source Java programs under active development [Blackburn et al., 2006a]. In Sections 3.4 and 3.7, we use `lusearch`, a search application from the industrial-strength Lucene framework. The `lusearch` benchmark behaves similar to commercial web search engines [Haque et al., 2015]. We confirmed that profiling findings for `lusearch` generalize to other DaCapo benchmarks.

Hardware & OS We use a 3.4 GHz Intel i7-4700 Haswell processor [Intel, 2013a] with 4 Chip Multiprocessor (CMP) cores, each with 2 way Simultaneous Multithreading (SMT) for 8 hardware contexts; Turbo Boost maximum frequency is 3.9 Ghz, 84 W TDP; 8 GB memory, 8 MB shared L3, four 256 KB shared L2s, and four private 32 KB L1 data caches, 32 KB L1 instruction caches, and 1.5 K μop caches for each core.

We use Linux kernel version 3.17.0 with the perf subsystem to access the hardware performance counters. We add to Linux a software signal that identifies threads, allowing thread switches to be identified by Shim on SMT.

3.6 Evaluation

This section evaluates the strengths, limitations, and overheads of a variety of Shim configurations and sampling rates. We start with simple, but realistic profiling scenarios, and build up more sophisticated ones that correlate software and hardware events. (1) We first compare Shim sampling a highly mutating software signal that stores loop and method identifiers in a single memory location on the same core in

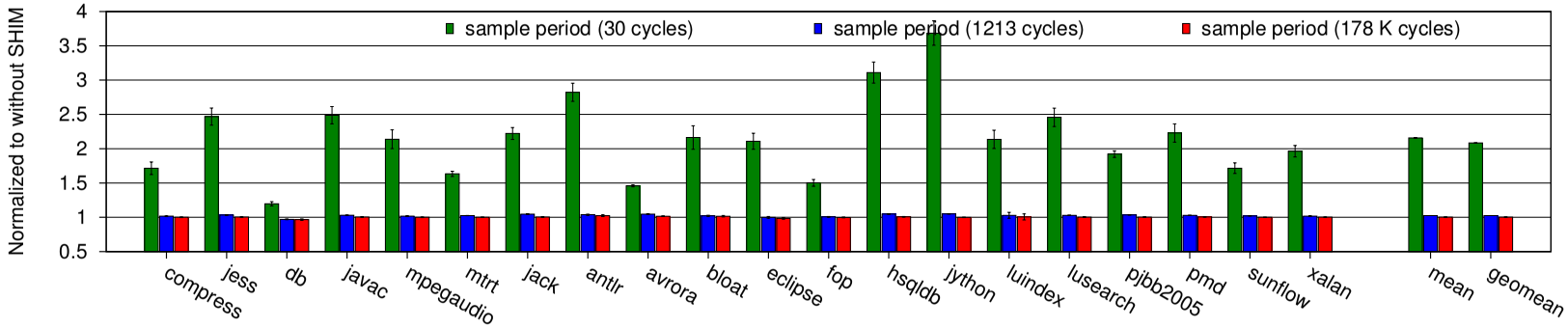
an SMT context and on a different CMP core. Both exhibit high overheads at very fine (<30 cycle) resolutions due to execution resource competition on SMT (~60%) and caching effects on CMP (~100%). However, CMP overheads are negligible for coarser (~1200 cycle) resolutions. (2) When SHIM computes IPC, a rate metric, on SMT with hardware performance counters (the relevant counters are not accessible on another CMP core), high frequency sampling overheads are 47%, similar to sampling a software signal on SMT. (3) We then configure SHIM to correlate method and loop identifiers with IPC on SMT and show that the overheads remain similar. (4) Finally, we show that if hardware vendors made local performance counters visible to the other cores, SHIM overhead on CMP for correlating IPC with a highly mutating software signal would drop to essentially nothing at a resolution of ~1200 cycles.

3.6.1 Observing Software Signals

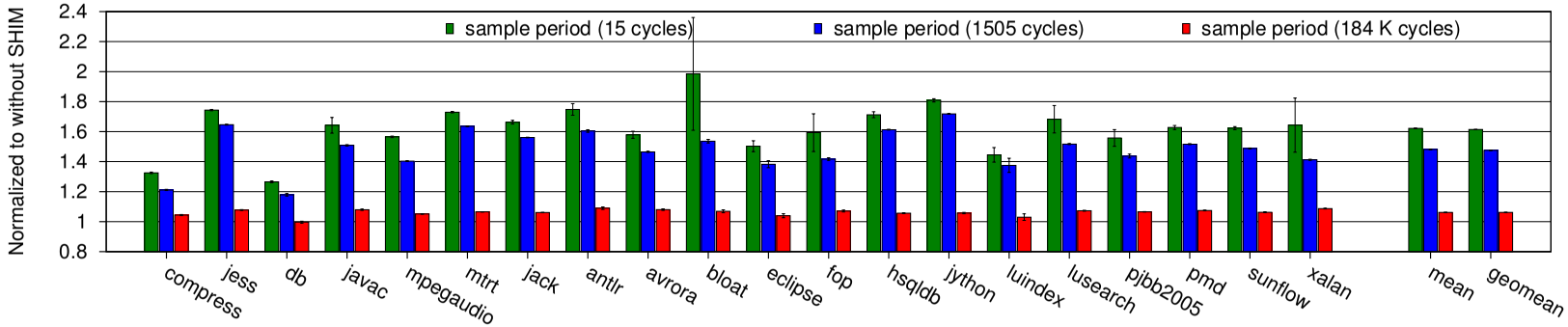
This section evaluates SHIM overheads in configurations on the same core in an SMT hardware context and on a separate core in a CMP hardware context when it samples a software signal by simply reading a memory location. Comparing these configurations shows the effects of sharing execution resources with SMT versus inducing cache traffic with CMP. We control sample rate and contention by idling the SHIM thread.

Method and loop identifiers We repurpose *yield points* to identify methods and loops as a fine-grain software signal. JVMs use yield points to synchronize threads for activities such as garbage collection and locking. The compiler injects yield points into every method prologue, epilogue, and loop back edge. A very efficient yield point implementation performs a single write to a guard page [Lin et al., 2015]. When the VM needs to synchronize threads, it protects the guard page, causing all threads to yield as they fault on their next write to the protected page. Jikes RVM implements yield points with an explicit test rather than a guard page, so for this experiment we simply add to each yield point a write of its method or loop identifier, adding an average of 1% execution time overhead. We measure yield point frequency and find that the average period ranges from 14 (jython) to 140 (avrora) cycles. The evaluation uses a JVM configuration with the additional store, but without SHIM, as the baseline. SHIM increments a bucket in a method-and-loop-identifier histogram. We pre-size this histogram based on an earlier profile execution. These changes produce a low-cost, high-frequency software signal that identifies the fine-grain execution state.

Overheads Figures 3.7(a) and (b) show SHIM sampling yield points with three sampling periods from 15 cycles to 184 K cycles on CMP and SMT hardware. A period of 184 K cycles (18.5 KHz) approximates the sample rate of tools such as VTune and Linux perf (1-100 KHz). We throttle SHIM to slow its sampling rate using `nop` instructions or by calling `clock_nanosleep()`, depending on the rate. The error bars on these and other figures report 95% confidence interval.



(a) SHIM CMP overheads are ~100% at a 300 cycle period, but drop at a 1213 cycle sample period to only 2%.



(b) SHIM SMT overheads range from 61% to 26%.

Figure 3.7: SHIM observing on SMT and CMP method and loop identifiers—a highly mutating software signal

The green bars in Figure 3.7(a) show Shim in a CMP configuration sampling a memory location as fast as it can, resulting in an average sample period of 30 cycles over all the benchmarks, incurring an overhead of around 100%. This sampling frequency imposes a high overhead on applications because each application frequently writes the yield point value, which dirties the line. Every time Shim's observer thread reads it, the subsequent write by the application must invalidate the line in the observer's cache. This invalidation stalls the application frequently at high sample rates. Section 3.6.2 examines this effect in more detail. Decreasing the sample period to ~1200 cycles reduces these invalidations sufficiently to eliminate most of the performance overhead.

The cost of observing software signals would be substantially reduced by a special read instruction that returns the value without installing the line in the local cache and where the cache coherence protocol is modified to ignore the read, as implemented in some ARM processors [Stevens, 2013].

Figure 3.7(b) shows the cost of observing a single software signal on the same core with SMT as a function of sampling rate. On the same core, Shim can sample memory locations at a higher rate compared to sampling from another core (every 15 vs 30 cycles). The rate on SMT compared to CMP is faster because Shim on SMT is not limited by cache coherence traffic, only by competition for execution resources. Note that restricting the observer thread to fewer CPU resources, for example one issue slot, using priorities (such as those on MIPS and the IBM Power series [Snavely et al., 2002; Boneti et al., 2008]) or some other mechanism, could significantly reduce this overhead.

Comparing the two, note that because the memory location mutates frequently, it is cheaper to sample on SMT than CMP at the highest sampling rates, but SMT is still relatively expensive for a ~15 cycle period, at 61%. However, with sample periods as low as ~1500 cycles, SMT overheads remain high, whereas CMP sampling overhead drops to a negligible amount. The next section studies these effects in more detail.

3.6.2 Software Signal Breakdown Analysis

Software signal overhead has two components: (1) application instrumentation, and (2) the Shim observer thread competing either for the cache line on a separate CMP core or for hardware resources on the same SMT core.

We use the microbenchmark in Figure 3.8 to understand how the rates of producing software signals and consuming them impacts overheads. The producer on the left generates events by writing into one memory location. The write to the local variable `dummy` forces the machine to make the write to `flag` architecturally visible. On the right, the consumer reads the `flag` memory location and increments a counter.

The inner `for` loops control producer and consumer rates, which we adjust by varying the number of `rdtscp` instructions. In Figure 3.9(a), the blue line (increasing p-wait) shows the consumer observing as fast as it can, while the producer slows its rate of event production on the x-axis. Conversely, the red line shows the producer writing as fast as possible, while the consumer slows its rate of consumption on the x-axis.

```
1  extern int flag;
2  void  producer() {
3    int dummy;
4    for(j=0; j<100000; j++){
5      for (i=0; i < p_wait; i++){
6        rdtsc();
7      }
8      write flag;
9      // force visibility of
10     // write to flag
11     write dummy;
12 }  }
```

(a) Producer

```
1  extern int flag;
2  extern int counter;
3
4  void consumer() {
5    while(1){
6      for(i=0; i< c_wait; i++){
7        rdtsc();
8      }
9      read flag;
10     counter++;
11   }
12 }
```

(b) Consumer

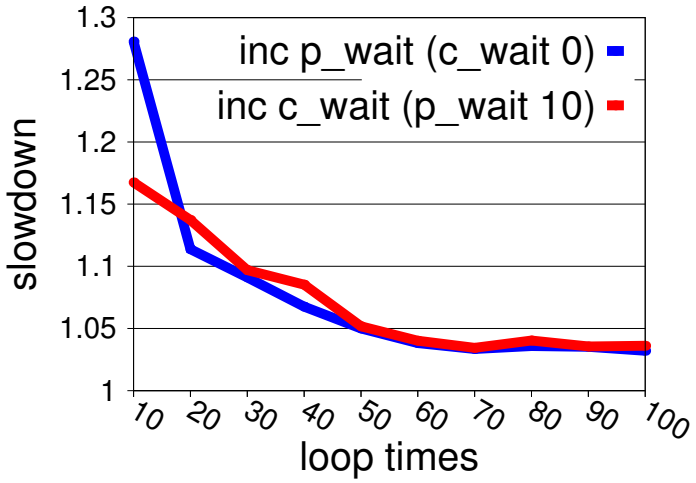
Figure 3.8: Microbenchmarks explore software overheads.

Except when both operate at their highest rate, both rates impact overhead similarly. Figure 3.9(b) reports the number of write-invalidations as a function of consumer (read) sample rate when the producer writes most frequently. More samples induce more invalidations. With highest frequency production and consumption rates, write-invalidation traffic dominates overhead, inducing observer effects. Increasing the sampling or production period drops overheads to less than 5%.

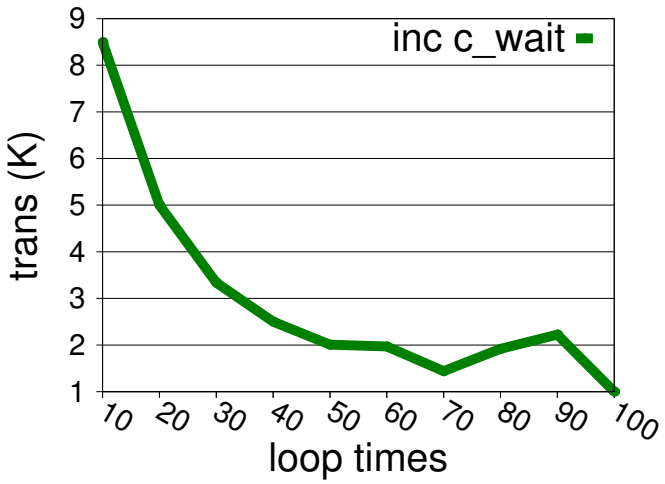
Figure 3.10 shows the same experiment on SMT hardware where overhead is dominated by the consumer because the critical CPU core resources are shared between two active SMT contexts. The more often the consumer reads the tag, the more it interferes with the producer. When the consumer is sampling as fast as it can, and the producer is writing at a high rate (the blue line in Figure 3.10(a)), they compete for shared execution resources. When the producer and the consumer increase the sampling or production period, the overhead drops to 10%, but much higher than the same CMP case (5%). The reason is that some critical CPU resources of the shared core are equally partitioned between two active SMT contexts, thus even the consumer samples at low frequencies, the producer is not able to use resources in the other partition.

3.6.3 Observing Hardware Signals

Figure 3.11 illustrates the overheads of Shim observing IPC, a rate-based hardware signal, on SMT at three sample rates. IPC cannot be evaluated directly on CMP because the instructions retired performance counter is not visible to other cores. In this experiment, Shim reads two retired instruction counters (one for the core, one for Shim itself), reads the cycle counter, computes application IPC and CPC, performs DTE, and builds an IPC histogram with 500 buckets for IPC values from 0 to 4. Because Shim consumes execution resources, it incurs overhead of around 47% at sample periods of ~400 and ~1900 cycles. Sampling every 185 K cycles incurs a penalty of 6.3%. Although overhead is relatively high, because we discard perturbed samples and the Shim observer thread offers a constant load, we believe that the signals are not obscured (recall the analysis in Section 3.4.3 and of Figure 3.6). Hardware manufacturers could validate our results with ground truth using their proprietary

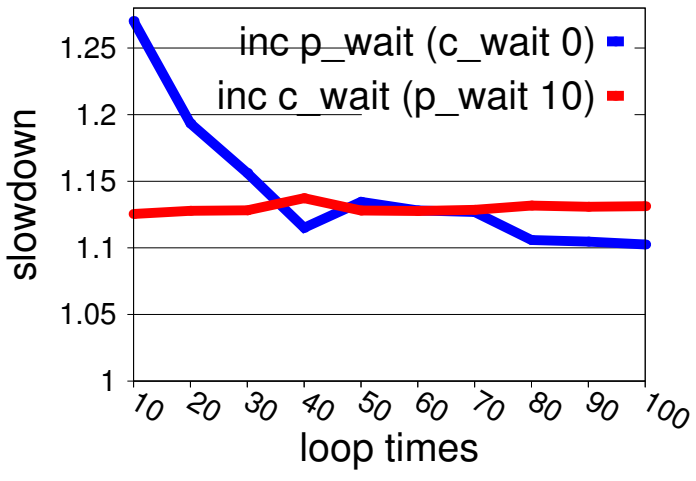


(a) Overheads as a function of producer and consumer rates.



(b) Producer generated write-invalidations as a function of consumer sampling rate.

Figure 3.9: CMP overheads. Write-invalidates induce observer effects and overheads. Increasing the producer or consumer periods drop the overheads to $< 5\%$.



(a) Producer is degraded by high consumption rate.

Figure 3.10: SMT overheads. SMT observer effects are highest as a function of producer, but then relatively constant at ~10%.

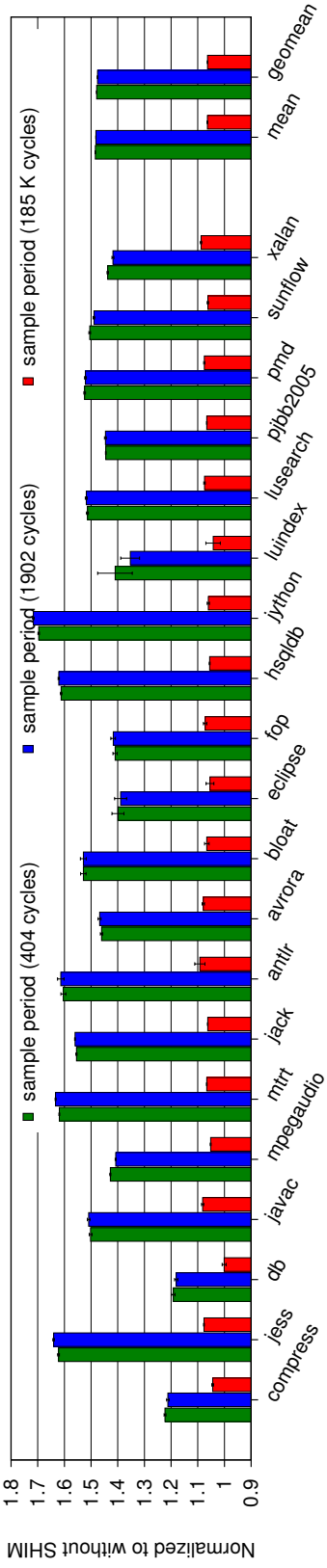


Figure 3.11: SHIM on SMT observing IPC as a function of sample rate. Overheads range from 47% to 19%.

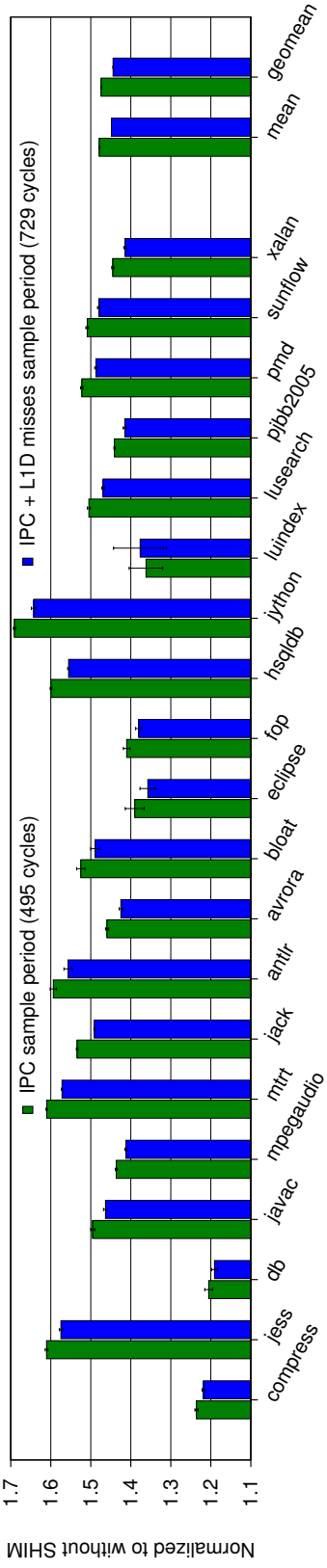


Figure 3.12: SHIM on SMT correlating method and loop identifiers with IPC and cache misses.

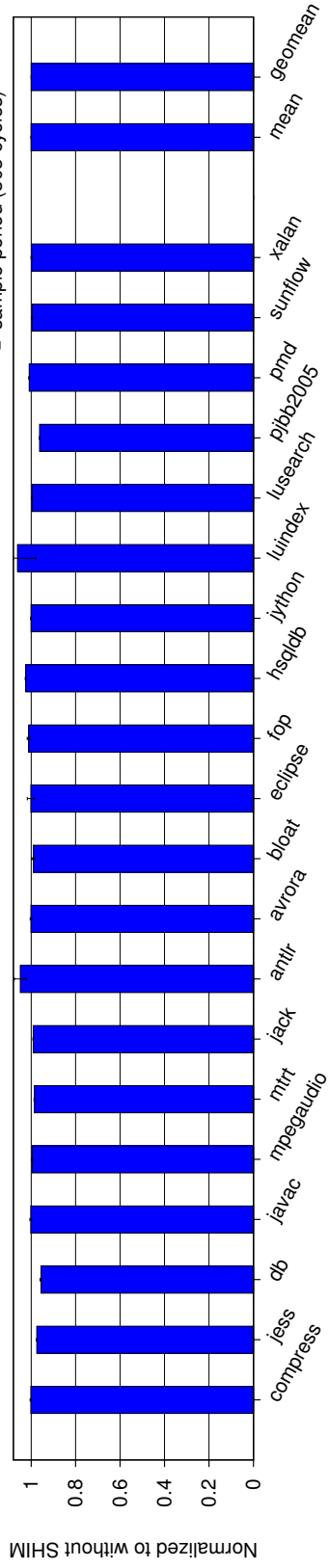


Figure 3.13: The overhead of SHIM is essentially zero when observing IPC from a remote core.

hardware measurement tools.

3.6.4 Correlating Hardware and Software Signals

This experiment measures overheads when we configure SHIM to correlate method and loop identifiers with IPC and with data cache misses. These are practical configurations that will help developers identify poorly performing low IPC loops and methods, and whether cache misses are responsible. Because SHIM needs two performance counters to correctly compute rate metrics, the cache miss configuration consumes five hardware performance counters.

Figure 3.12 compares SHIM sampling as fast as it can when it samples method and loop identifiers and IPC, with sampling them plus cache misses. Adding another performance counter makes SHIM sample more slowly (729 versus 495 cycles) because it must read both core and SHIM counters and it is limited by the 30 to 40 cycle latency of reading each counter and executing a `rdtscp` instruction. However, slowing SHIM to gather more hardware information incurs less overhead because it stalls more often consuming less shared CPU resources, inducing fewer observer effects on the application. Section 3.7 shows two detailed case studies on critical methods using similar configurations that reveal how this fine-grain information generates hardware and software insights that prior work cannot and that suggest future directions for optimizations and mechanisms.

3.6.5 Negligible Overhead Fine-Grain Profiling

This section shows that if all profiling work could be performed on the separate CMP core, overheads and observer effects would be extremely low. Figure 3.13 shows SHIM reading three hardware performance counters on a separate CMP core, sampling as fast as it can, which results in a period of ~300 cycles on average. We use three counters because this is the minimum required to compute a rate. The time to read one performance counter ranges from 30 to 40 cycles, limiting the sample rate. SHIM executes the reads in sequence with the synchronous `rdtscp()` instruction (line 6 of Figure 3.2), because all reads must be performed in order to correctly correlate and count events. This analysis shows that SHIM can operate at a high sample rate with no statistically significant overhead when reading hardware signals from another core. This result motivates increasing the visibility of core-private hardware performance counters to other cores.

3.7 Case Studies

This section shows examples of the diagnostic power of fine-grain program observations and compares it to the *average* results (reported in the top of each figure in this section) that previous tools must report for these same metrics because their sampling period is much longer.

We consider two phases of performance-critical garbage collection in Jikes RVM. We first examine the response of the two phases when using DVFS to change the frequency from 3.4 to 0.8 GHz. Then we examine their response to turning on and off the hardware prefetcher. We instrument the collector with software signals that identify the phases. Shim reads the software and hardware signals to compute IPC and memory bandwidth and attributes them to the appropriate phase.

We choose two phases on the critical path of garbage collection: (1) *stack scanning* (stacks), where the collector walks the stacks of each executing thread, identifying all references and placing them in a buffer for later processing, and (2) *global scanning* (globals), where the collector walks a large table that contains all global (static) variables, identifies references and places each reference in a buffer. Superficially, the phases are similar: the collector walks large pieces of contiguous memory identifying and buffering references which it processes in a later phase. In the case of globals, a single contiguous byte map straightforwardly identifies the location of each reference. Global scanning performs a simple linear scan of the map. On the other hand, stack scanning requires unwinding each stack one frame at a time and dereferencing a context-specific stack map for every frame to find the references within it. This highly irregular behavior leads to poor locality.

In a modern generational garbage collector, these phases can dominate the critical path of frequent 'nursery' collections, particularly in the frequent case where object survival is low. Therefore, VM developers are concerned with their performance. The good and poor locality of these two phases also serves as a pedagogical convenience for our analysis of DVFS and prefetching.

3.7.1 DVFS of Garbage Collection Phases

This section evaluates IPC and memory bandwidth at two clock speeds: 3.4 GHz (default) and 0.8 GHz on the Haswell processor. Figure 3.14 plots IPC and memory bandwidth for the stack phase (poor locality) and Figure 3.15 plots the global phase (good locality). Figure 3.14(a) plots the distribution of sampled IPC values at 3.4 GHz (purple) and 0.8 GHz (orange). The slower clock improves the IPC from 0.59 to 0.83, which is unsurprising for a memory-bound workload because memory accesses are relatively lower latency at lower clock rates. The purple line shows a large spike where many samples observe an IPC of ~0.10 at 3.4 GHz. This spike disappears when at 0.8 GHz (orange line), instead the IPC distribution is quite uniform. However, the slower clock speed has almost no affect on the distribution of samples above 1.3, which presumably reflect program points that are not memory bound.

Figure 3.14(b) shows the difference in memory bandwidth consumption between the two clock rates on the stacks phase. The histograms bucket samples according to IPC (x-axis) and for each bucket plots the average number of memory requests per 100 cycles for 3.4 GHz (purple) and 0.8GHz (orange). The lower clock rate increases memory bandwidth by 20% from 0.91 to 1.09 memory requests per 100 cycles. When IPC is low, the memory bandwidth increases by a factor of two from about 1.4 requests per 100 cycles to about 3.

The memory-bound stack phase may benefit from a DVFS-reduced clock rate because the relatively more effective use of memory bandwidth leads to a 40% improvement in IPC. This fine-grained analysis also shows that the phase is not homogenous, and many samples show little response to DVFS.

Figure 3.15(a) plots the distribution of sampled IPC values for the globals phase (good locality) at 3.4 GHz (purple) and 0.8 GHz (orange). The graph shows a strikingly more focussed and homogenous distribution than the stacks phase. Interestingly, we see a counter-intuitive IPC *reduction* for the lower clock speed. Figure 3.15(b) shows that there is no clear change in memory bandwidth. The data to the left of Figure 3.15(b) is very noisy, but this noise is due to a paucity of samples—95% of all DRAM requests are due to samples with IPCs greater than 1.2. A slightly lower IPC at a slower clock is non-intuitive, but we hypothesized that the hardware prefetcher was responsible and examine this hypothesis next.

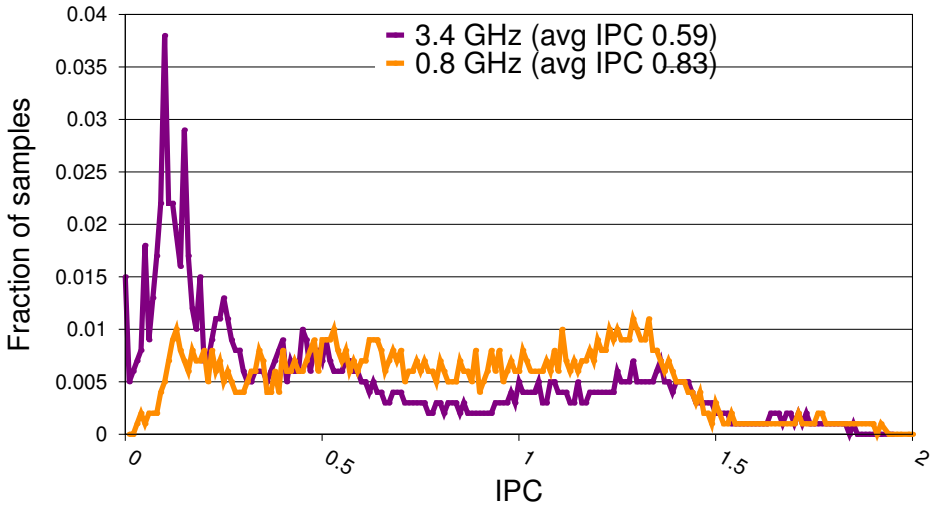
Before we continue, note that fine-grain sampling reveals the unique behavior of global scanning in the context of the entire garbage collection, which is memory bound on average and thus more resembles stack scanning. As the granularity of sampling increases it will tend toward the average for the whole of garbage collection, obscuring the distinct behavior of globals and stacks. Even if the two behaviors were equally representative of garbage collection, coarse-grain sampling may still miss the drop in IPC, because the magnitude of the response to DVFS is so much higher for the stacks.

This section compares the effect of enabling and disabling the hardware prefetcher on IPC and memory bandwidth on the two phases. Figure 3.16 plots the stacks (poor locality). Figure 3.16(a) plots the distribution of sampled IPC values with (purple) and without (orange) prefetching. The differences are modest; on average turning off the prefetcher reduces IPC from 0.60 to 0.57; a 5% reduction. On the other hand, Figure 3.16(b) shows a more substantial reduction in memory bandwidth, from 0.91 requests per 100 cycles to 0.68; a 25% reduction. Together these graphs suggest that the hardware prefetcher is not effective at compensating for poor locality since the reduction in memory traffic outstrips the IPC decrease by a factor of 5.

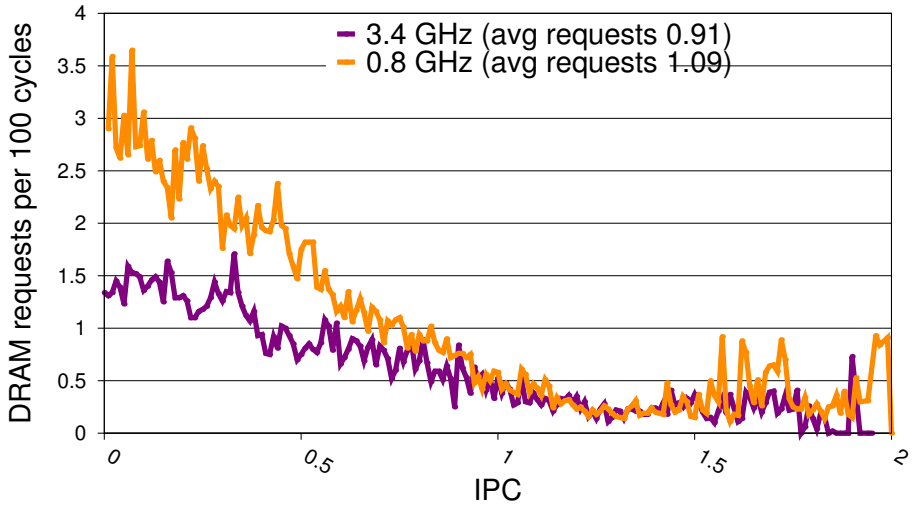
Figure 3.17 considers the hardware prefetcher and the globals phase (good locality). Figure 3.17(a) shows a very clear reduction in IPC when the prefetcher is disabled, from 1.48 to 1.15. Figure 3.17(b) shows that unlike the stack roots phase, the average memory bandwidth is unaffected (values less than 1.2 IPC with prefetching on contribute only 5% of traffic and are noisy due to a paucity of samples).

3.7.2 Hardware Prefetching of Garbage Collection Phases

In the stack phase (poor locality), the prefetcher is not effective and it consumes additional memory bandwidth compared to not prefetching at all, reshaping the data in the caches, and to no effect. Whereas the prefetcher for the globals (good spatial locality) is so accurate that it not only delivers the correct data in a timely fashion, it actually reduces memory bandwidth. These results suggest that if hardware vendors provide low latency ways to adjust DVFS and prefetching, a dynamic

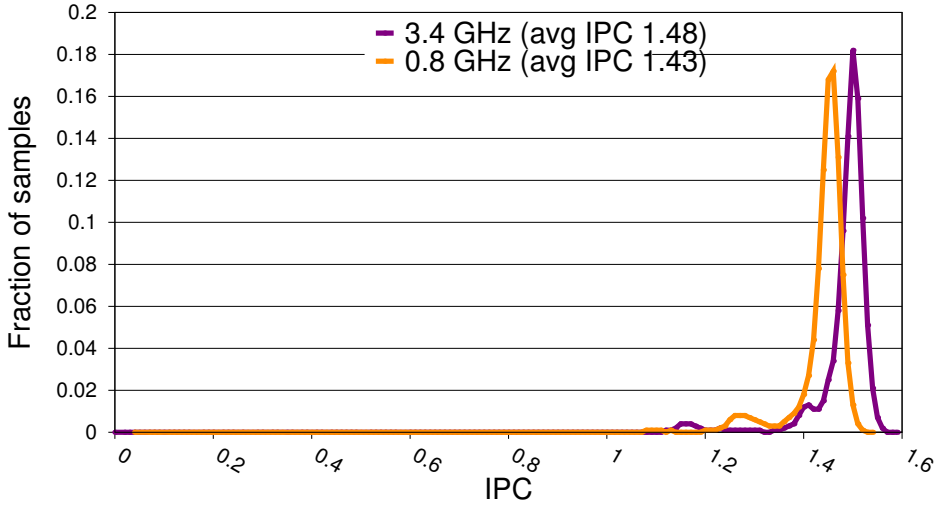


(a) IPC frequency distribution

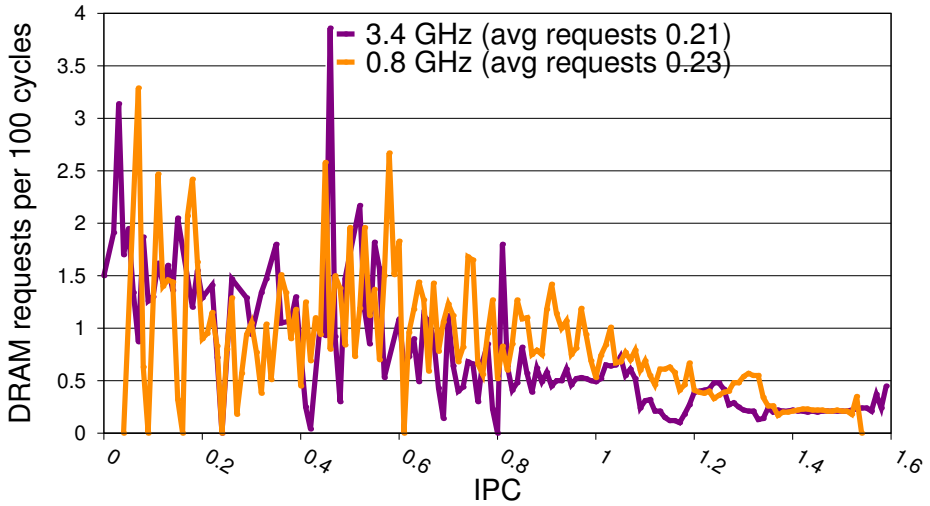


(b) Average memory bandwidth relative to IPC

Figure 3.14: DVFS effect at 3.4 and 0.8 GHz on IPC and memory bandwidth for stacks (poor locality).

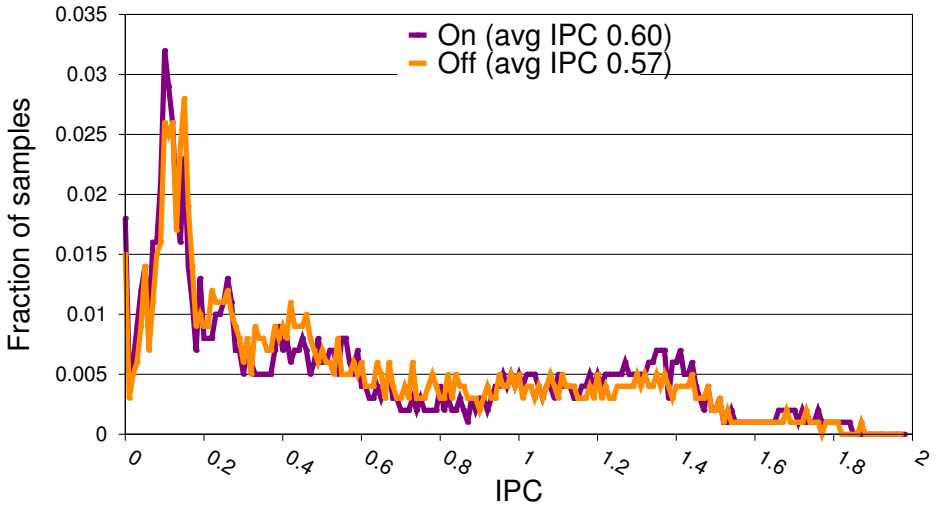


(a) IPC frequency distribution

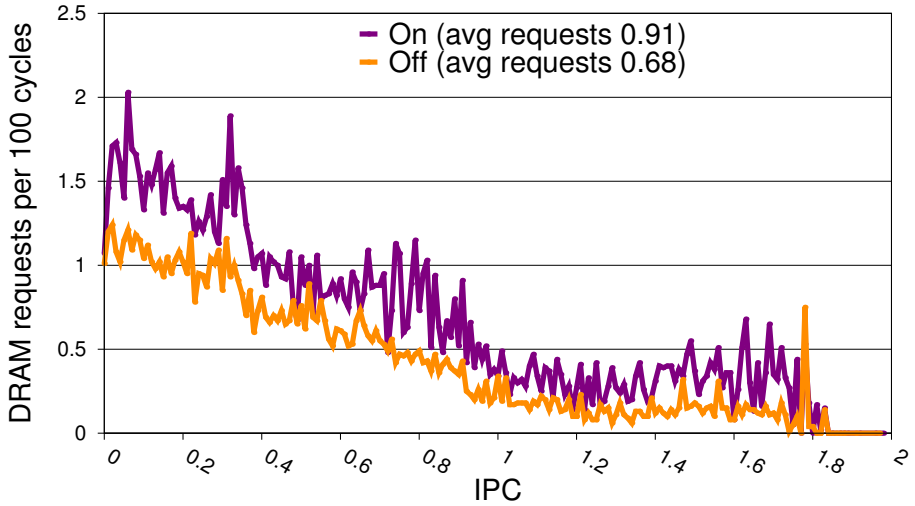


(b) Average memory bandwidth relative to IPC

Figure 3.15: DVFS effect at 3.4 and 0.8 GHz on IPC and memory bandwidth for globals (good locality).

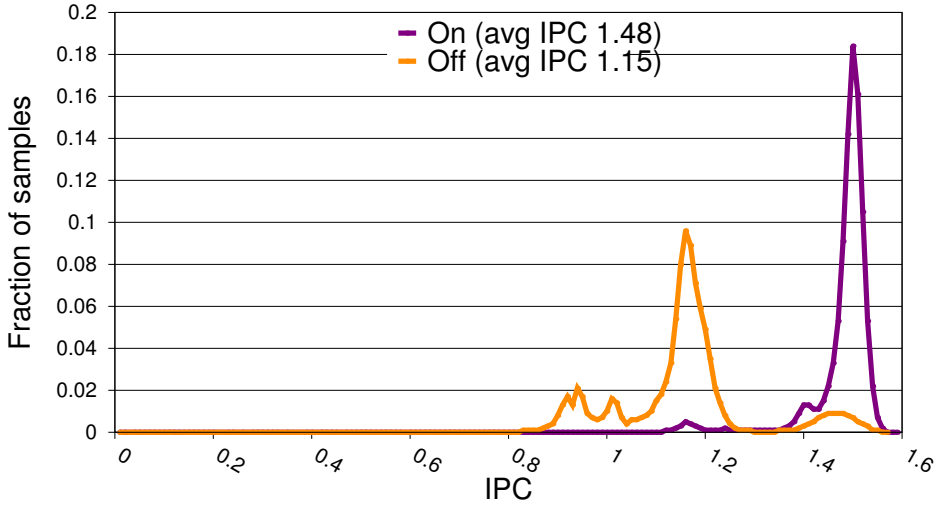


(a) IPC frequency distribution

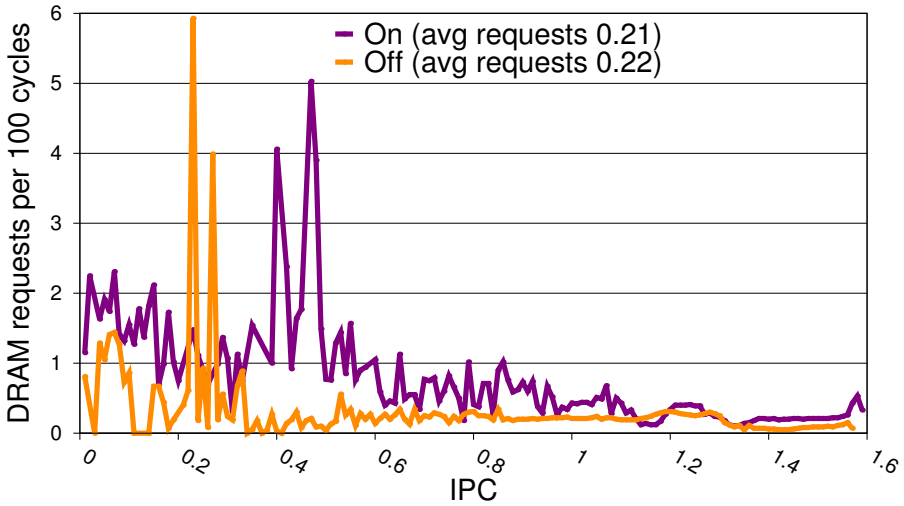


(b) Average memory bandwidth relative to IPC

Figure 3.16: Prefetching effect (on/off) on IPC and memory bandwidth for stacks (poor locality).



(a) IPC frequency distribution



(b) Memory bandwidth relative to IPC

Figure 3.17: Prefetching effect (on/off) on IPC and memory bandwidth for globals (good locality).

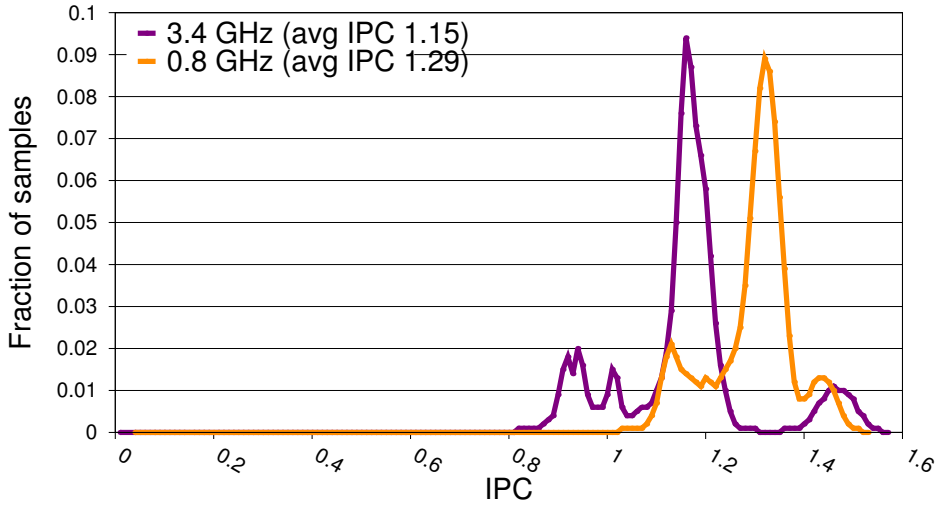


Figure 3.18: DVFS effect on IPC for globals with prefetching off.

optimization could improve efficiency and perhaps performance by adjusting DVFS and prefetching at a fine granularity.

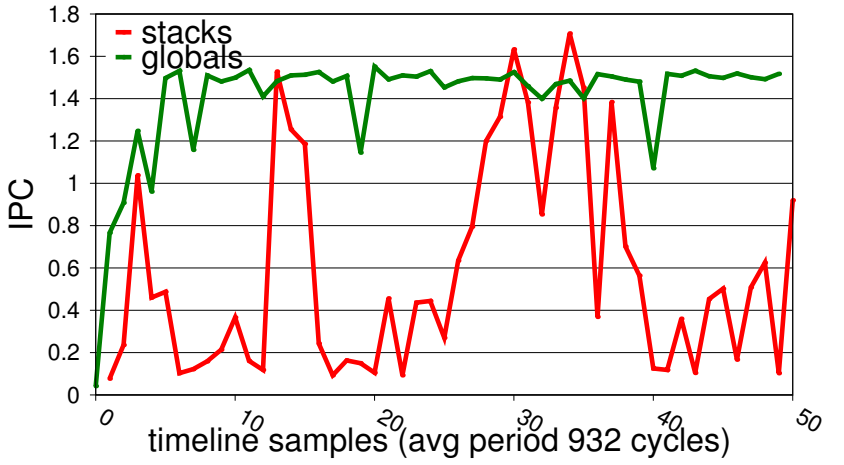
Figure 3.18 reconsiders the effect of DVFS on globals, this time with prefetching disabled. We find that IPC increases from 1.15 to 1.29, matching intuition and confirming our hypothesis (compare to Figure 3.15(a)).

Figure 3.19 plots time line series for IPC and memory bandwidth. These figures further illustrate the different IPC behaviors of the two phases and that memory bandwidth consumption is highly correlated with low IPC, explaining their behaviors. Note that simply examining the averages in these results that coarse-grain tools would produce does not lend itself to these insights.

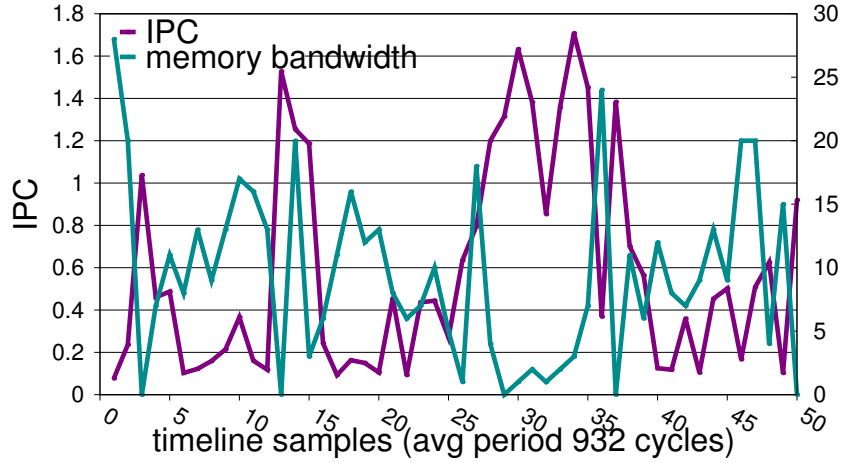
3.8 Related Work

Four themes in profiling and performance analysis are most related to our work: profilers that sample application behavior using interrupts, instrumentation, direct performance measurement, simulators and emulators, and feedback-directed optimization.

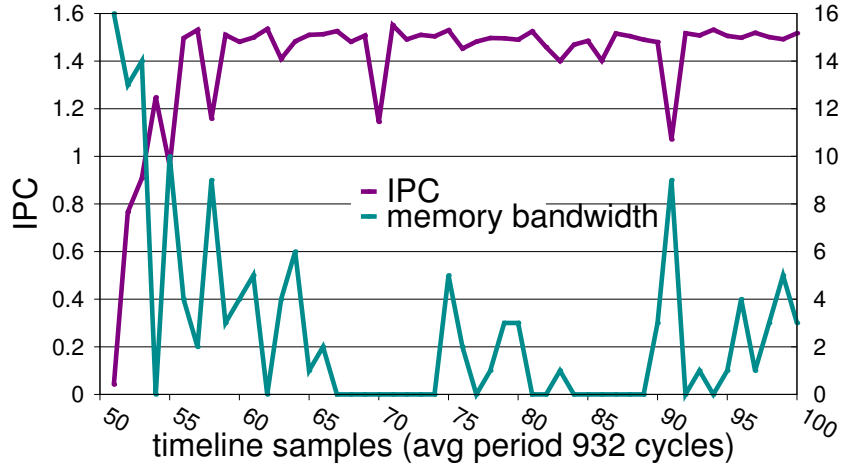
Interrupt-Driven Sampling Interrupt-driven samplers have proved invaluable at helping systems builders and application writers by performing low overhead sampling, identifying software hotspots, and attributing performance pathologies to code locations. DCPI [Anderson et al., 1997] and Morph [Zhang et al., 1997] are progenitors of today's profiling tools such as VTune, OProfile, Linux perf, and top-down analysis [Intel, 2014; Linux, 2014a; OProfile, 2014; Strong, 2014; Yasin, 2014]. These systems use interrupts to sample system state and build a profile of software/hardware interactions. DCPI introduced random sample intervals to avoid the sampling



(a) IPC for 50 consecutive samples



(b) stacks



(c) gplobals

Figure 3.19: Strong correlation between IPC and memory bandwidth revealed in time line series for stacks and globals.

bias suffered by prior timer-based systems [Anderson et al., 1997]. Elapsed time or a count of hardware events may define the sample period. Nowak and Bitzes [2014] overview and thoroughly evaluate Linux perf on contemporary hardware. New hardware and software support, such as Yasin's top-down analysis [Yasin, 2014], PEBS, and *Processor Tracing* [Strong, 2014] add rich information about the hardware and software context at each sample, but are still limited by sample rate.

Because the OS services interrupts on critical code paths within the kernel, interrupt driven systems must throttle their sample rate to avoid system lockup [Anderson et al., 1997; Intel, 2014; Linux, 2014b]. The dependence on interrupts limits these tools' ability to resolve performance behavior to events visible at sampling periods of around 30 K cycles. In contrast, Shim exploits unutilized hardware contexts to continuously monitor performance, instead of using interrupts, and thus operates at resolutions as fine as 15 cycles. At a period of around 1 K cycles, many configurations have very low overhead. Shim avoids sample bias as a result of its natural variation in sample period.

Direct Measurement Directly reading hardware performance counters is also a widely used approach [Ammons et al., 1997; Pettersson, 2003; Linux, 2014a; Demme and Sethumadhavan, 2011; Zhao et al., 2008; Ha et al., 2009]. Unfortunately perturbing the code with performance counter reads that take ~30 to 40 cycles each induces observer effects–shorter periods increase coverage, but increase observer effects. In contrast, Shim reads hardware counters without perturbing the application code itself.

Instrumentation Profilers Instrumentation profiling tools such as PiPA and CAB insert code into applications that records software events of interest in a buffer, such as paths and method identifiers, for online or offline processing [Ha et al., 2009; Zhao et al., 2008]. They however did not correlate hardware and software events, although their frameworks could support it. Ammons et al. [1997] combine path profiling with hardware events, but suffer substantial performance overheads. Shim profiles software events at very low overhead, either by injecting code that emits software signals, or by observing existing software signals. Instrumentation profilers such as PiPA and CAB can produce complete traces or samples. Sampling profilers such as Shim cannot guarantee a complete trace, so are unsuitable when completeness is a requirement. Mytkowicz et al. show that despite the problem being identified a decade earlier [Anderson et al., 1997], many instrumentation profilers still suffer bias [Mytkowicz et al., 2010].

Simulators and Emulators Simulators and emulators profile software at instruction and even cycle resolution. Shade [Cmelik and Keppel, 1994], Valgrind [Nethercote and Seward, 2007], and PIN [Luk et al., 2005; Wallace and Hazelwood, 2007] are examples of popular tools that use binary re-writing and/or interpretation to instrument and profile application code. Although they profile at an instruction level and emulate

unimplemented hardware features, these tools are heavyweight. Instead of measuring hardware performance counters, they instrument code and emulate hardware. They are therefore unsuitable for correlating fine-grain hardware and software events.

Feedback-Directed Optimization Feedback-directed optimization is an essential element of systems that dynamically compile, including managed language runtimes and dynamic code translation software such as Transmeta's code morphing software [Dehnert et al., 2003]. These systems use periodic sampling to identify and then target hot code for aggressive dynamic optimization. SHIM provides a high resolution, low overhead profiling mechanism that lends itself to feedback directed optimization.

3.9 Summary

Performance analysis is a critical part of computer system design and use. To optimize systems, we need tools that observe systems at granularities that can reveal their behavior. SHIM views computer systems as hardware and software signal generators. It repurposes existing hardware to execute a profiling observer thread that simply reads performance counters and memory locations to sample hardware and software signals. We show that configurations of SHIM offer a range of overheads, sampling frequencies, and observer effects. We show how to correct for noise and control for some observer effects. We propose modest hardware changes that would further reduce SHIM's overheads and observer effects. We present case studies that demonstrate how this performance microscope delivers a new level of analysis and potential optimizations.

Engineers can observe the fine-grain behavior of online production systems with SHIM. However, to design new optimizations, having the power of seeing behavior is not enough, they need to analyze and control system behavior such as fine-grain interactions between system components. The next chapter introduces a SHIM-based real-time latency controller, TAILOR, that continuously monitors hazardous system behaviors and takes real-time control actions to reduce their impact.

Chapter 4

Tailor

Engineers craft computer systems by using and providing rich abstractions. The result is a complex system stack. This approach improves engineer productivity significantly, but also poses a challenge to latency-critical web services since these services have to consistently deliver fast responses. This is in part because behaviors of system components such as slow wake-ups and garbage collections can delay latency-critical requests by orders of magnitude longer than their target latency. It is hard to identify and address such hazardous behaviors because they are rare, unpredictable, invisible, and may even be unavoidable.

The previous chapter introduced Shim, a new profiling approach, that can observe fine-grain system behavior. Shim's high-frequency, high-fidelity profiling enables a new approach to monitoring and controlling hazardous system behaviors for latency-critical web services. This chapter introduces Tailor, a real-time latency controller that uses a Shim-based high-frequency profiler and an application-level network proxy to continuously monitor and act on hazardous system behaviors. Tailor identifies such behaviors by recording and analyzing fine-grain interactions of system components. For web services that support local-node redundancy, Tailor uses a backup server on the same machine to mitigate the impact of unavoidable random hazardous behaviors.

This chapter is structured as follows. Section 4.2 describes the design and implementation of Tailor. Section 4.3 shows our experimental setup, and Section 4.4 demonstrates how Tailor identifies and addresses hazardous system behaviors.

4.1 Introduction

Developers of modern latency-critical web services, from search to key-value store services to server-side rendering, leverage the power of high-level abstractions, exploiting managed languages, third-party libraries, application frameworks, and operating system services. The result is a complex system stack. For example, Hypernova [Airbnb, 2018], AirBnB's server-side rendering service implemented in JavaScript and running on the Node.js run-time system, relies on the React library [Facebook, 2018] to render web pages. ElasticSearch [Elastic, 2019], a Java distributed search engine, delegates core indexing and searching work to Apache Lucene [Apache Lucene, 2014], a Java

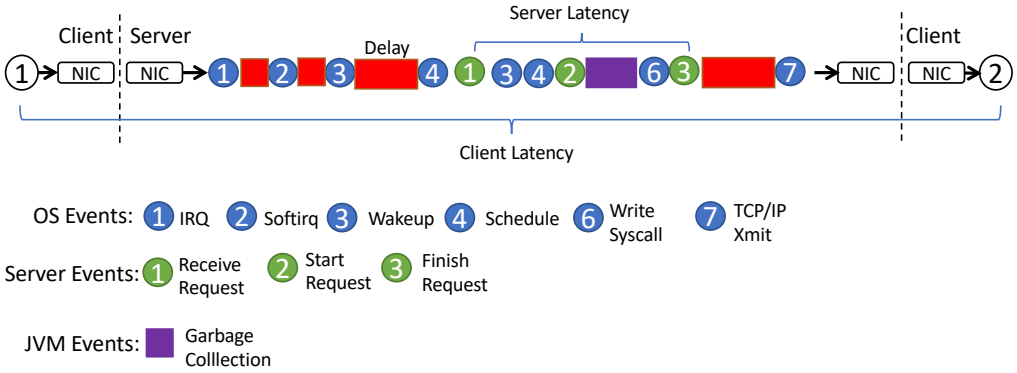


Figure 4.1: The life of a latency-critical request. Notice that the client-observed latency (bottom) is very different to the server-observed latency (top).

open-source information retrieval software library. And beneath this stack must sit an operating system with built-in network support and all the complexity it brings.

At the same time, to meet user performance requirements, interactive web services have strict service-level objectives (SLOs) on *tail latency*, for example, a 99th percentile latency of 10 ms. It is a challenge to consistently deliver low-latency responses on a complex system stack, in part, because system components can delay latency-critical requests by orders of magnitude larger than the target latency.

Figure 4.1 shows the life of a simple request. While the request travels through the system stack, many system behaviors can be hazardous to the request latency, for example, the kernel can delay scheduling to the application server after the wake-up (the red bar between the `wakeup` event and the `schedule` event); and the operating system network stack can delay data transmission (the red bar between the `finish-request` event and the `TCP/IP-xmit` event). Many of these delays caused by the operating system may occur before or after the application actually processes the request, thus compared with client-observed latency, the server-observed latency may differ greatly.

It is hard to identify such hazardous system behaviors because they are rare, unpredictable, and typically beyond the applications' control. Unfortunately, it is even harder to address these behaviors. One typical example is garbage collection, which provides a convenient memory usage abstraction to applications and runs periodically to recycle memory. Collections are rare and unpredictable. When they happen, they may take a significant portion of computation resources for a while. Some collections may even stop all application threads for a long period.

Shim's high-frequency profiling enables a new approach to identifying and addressing these hazardous system behaviors in real time. This chapter introduces Tailor, a real-time latency controller that uses a Shim-based high-frequency profiler and an application-level network proxy to monitor and act on hazardous system behaviors. Tailor continuously identifies hazardous system behaviors for slow requests by recording and analyzing request-related events generated by system components. It presents timelines showing how hazardous system behaviors affect slow requests

to help engineers understand and address root causes. For unavoidable random hazardous system behaviors, such as garbage collections, Tailor uses local-node redundancy to mitigate their impact.

To identify hazardous behaviors, Tailor creates software channels and a set of request-related events, such as wake-ups and scheduling in the OS, garbage collections in the JVM, and starting and finishing requests in the application server. Signals of these events show how the system stack processes requests at a fine granularity. Tailor instruments system components to generate these request-related events into the software channels. While the proxy forwards requests and responses for the active and backup servers, Tailor uses the high-frequency profiler to continuously record the events from the software channels to a timeline stream and also index the stream with the request identifier for speeding up analyzing. Whenever the proxy detects a slow request, it searches related events from the timeline stream, analyzes how system behaviors affect the request latency, and then presents a timeline view to help developers understand and address hazardous system behaviors.

However, some hazardous system behaviors are unavoidable, such as garbage collections. For web services that support local-node redundancy, Tailor uses a local redundant server to mitigate the impact of rare, random but unavoidable behaviors. The Shim profiler continuously detects pre-programed hazardous events. As soon as it detects one, it immediately takes real-time actions to reduce the event's impact. For example, to act on garbage collections that occur on the active server, Tailor switches the backup server with the active server, and then re-sends all on-going requests to the new active server which is unlikely to be doing a garbage collection.

In this chapter, we use a Lucene-based search server from the Lucene regression performance framework [McCandless, 2019] as our latency-critical service. We configure a set of search requests that demand less than 2.5 ms server time. We show that when sending these requests to the search server under light load on Linux, the requests can be delayed by as much as 46 ms. We demonstrate that Tailor identifies major root causes of the long tail: TCP buffering, slow page faults after the deep sleep, and unavoidable JVM pauses caused by class loading and garbage collections. We adjust system configurations to address the avoidable ones. For unavoidable JVM pauses, Tailor reduces their impact by scheduling in-flight requests to the backup server. We show that with these optimizations, Tailor reduces the maximum client latency by nine times from 41-46 ms to 3-5 ms.

In summary, contributions of this chapter include:

- Tailor, a real-time latency controller that continuously monitors and acts on hazardous system behaviors.
- A set of Shim events for tracking latency-critical network requests.
- The use of the local-node redundancy to mitigate the impact of random unavoidable hazardous behaviors.

4.2 Tailor Design and Implementation

This section describes the design and implementation of Tailor, in which a Shim profiler and an application-level network proxy are combined together to continuously monitor and act on hazardous system behaviors for latency-critical web services. Tailor instruments system components to generate TaleChain events that show how the system stack processes latency-critical requests into software channels with low overhead. Inside the Tailor process, a Shim observer thread continuously samples TaleChain signals at high frequencies on a separate hardware context, recording TaleChain events into a timeline stream and promptly taking preprogrammed actions on hazardous TaleChain events. Another proxy thread also runs on a separate hardware context, forwarding requests and responses for both active and backup servers, analyzing and presenting how system behaviors affect the latency of slow requests. On hardware with SMT support, Tailor binds the profiler and the proxy threads on the same CPU core but different hardware threads, saving hardware resources.

This section first explains TaleChain events, then describes the Tailor profiler and proxy, and finally shows how Tailor uses local-node redundancy to mitigate the impact of unavoidable JVM pauses by taking real-time control actions on JVM events.

4.2.1 TaleChain Events

As discussed in Section 2.1.1, we view computer systems as signal generators whose behavior can be observed from their software and hardware events. Tailor observes the behavior of latency-critical web services with TaleChain events.

Table 4.1 lists TaleChain events used for analyzing the behavior of Lucene search requests, covering four major components of the system stack: Linux, the JVM, Lucene servers, and Tailor. All events have shared fields indicating what the event is (the `type` field) , when the event is generated (the `timestamp` field), and who generates the event (the `tid` and `pid` fields). Events may also have other fields representing event-specific semantics, for example `to_tid` and `to_gid` of the `wakeup` event identify the target task that was woken up by the event generator.

TaleChain events have strong semantic connections. For example, a timeline of sequential events ranked by the `timestamp` reflects the behavior of system components in a period. The `timestamp` distance between a `wakeup` event and the following `schedule` event with the same `to_tid` and `to_gid` fields indicates how long the task waits in the scheduling queue. The timestamp distance between a `Systemcall` event and the following successful `TCP-xmit` event with the same buffer address shows how long the request response was buffered by the network stack.

Semantic connections between events enable Tailor to analyze how system behaviors affect requests. Figure 4.2 shows a timeline of sequential TaleChain events. The timestamp distance between the proxy-start event (yellow color) and the proxy-end event (yellow color) is the server-observable latency, and events in the period reflect how system components behave when processing the request. Following semantic

Event Type	**Event Source**	**Shared Fields**					**Event Fields**					
Wakeup	Kernel	timestamp	type	seq	tid	pid	to_tid	to_gid				
Schedule	Kernel	timestamp	type	seq	tid	pid	to_tid	to_gid	wakeup_tid	wakeup_pid	wakeup_timestamp	
Softirq	Kernel	timestamp	type	seq	tid	pid	id	latency				
TCP recv	Kernel	timestamp	type	seq	tid	pid	socket	sku_buffer	status			
TCP xmit	Kernel	timestamp	type	seq	tid	pid	socket	sku_buffer	status			
TCP ack	Kernel	timestamp	type	seq	tid	pid	socket	sku_buffer	status			
Pagefault	Kernel	timestamp	type	seq	tid	pid	fault_addr	latency				
Systemcalll	Kernel	timestamp	type	seq	tid	pid	number	latency	parameter1	parameter2	parameter3	
Receive a request	Proxy	timestamp	type	seq	tid	pid	queue_length	task				
Finish a request	Proxy	timestamp	type	seq	tid	pid	queue_length	task				
Receive a request	Lucene	timestamp	type	seq	tid	pid	queue_length	task				
Start a request	Lucene	timestamp	type	seq	tid	pid	queue_length	task				
Finish a request	Lucene	timestamp	type	seq	tid	pid	queue_length	task				
Yieldpoint	JVM	timestamp	type	seq	tid	pid	type	latency				

Table 4.1: TALECHAIN events for the Lucene search workload.

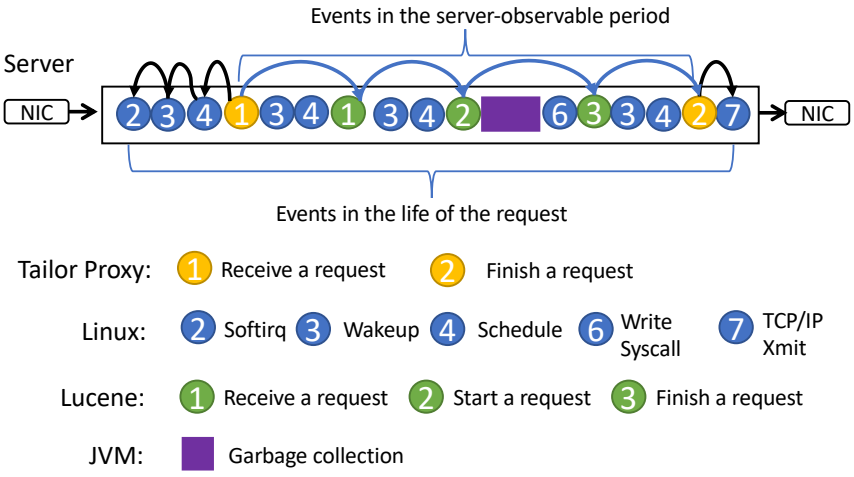


Figure 4.2: A timeline of TaleChain events shows the behavior of requests.

links between TaleChain events, Tailor is able to trace the request flow, as shown in Figure 4.2 where the blue arrows chain the proxy-start event, then the Lucene-receive event, the Lucene-start event, the Lucene-end, and finally the proxy-end event.

As discussed before, the server-observed latency can be wildly different from the client-observed latency since hazardous system behaviors may happen outside of the server-observable period. With TaleChain events, Tailor is able to trace from the proxy-start event back to the event corresponding to reading requests from ethernet card (`softirq`), and from the `proxy-end` event forward to the events corresponding to sending out the response to the ethernet card, presenting an end-to-end timeline view of requests. The black arrows in Figure 4.2 show the process of expanding the period by following semantic links: The `proxy-start` event is linked to the previous schedule event as the two events are generated by the same task, then back to the wakeup event that shows which task wakes up the proxy thread, finally back to the `softirq` event indicating when the operating system network stack processes the incoming TCP segments of the request; The `proxy-end` event is linked forward to the `TCP/IP-xmit` event showing when the network stack transmits the response to the ethernet card.

Tailor instruments system components to generate TaleChain events into software channels (memory locations). We express the type of TaleChain events as structures in C (line 9 in Figure 4.3), and the type of software channels as a union of event structures (line 1 in Figure 4.4). Software channels can be viewed as single-entry buffers where generating a new event to the same channel overwrites the old one, it is the profiler's job to record high-fidelity, high-frequency signals with the right set of counters. For example, for the generators that concurrently generate high-frequency events, the profiler can use per-CPU or per-task channels to reduce cache contention between the generators. For generators that mix high-frequency events with important low-frequency events, to avoid signal aliasing, the profiler can

```
1  struct shared_fields
2  {
3    u64 timestamp;
4    u32 type;
5    u32 seqid;
6    u32 tid;
7    u32 pid;
8  }
9  struct wakeup_event
10 {
11   struct shared_fields fields;
12   int to_tid;
13   int to_pid;
14 };
15
16 /******* signal generator ******/
17 core.c::try_to_wake_up(struct task_struct *target)
18 {
19   struct wakeup_event *channel;
20   ......
21   channel = __this_cpu_read(kernel_2nd_channel);
22   // shared fields
23   channel->type = WAKEUP_SIGNAL_TYPE;
24   channel->tid = current_tid;
25   channel->pid = current_pid
26   c->seqid += 1;
27   // wake-up specific fields
28   c->wakeup.to_tid = task_pid_nr(target);
29   c->wakeup.to_pid = task_tgid_nr(target);
30   // do this last
31   channel->timestamp = rdtsc();
32   .....
33 }
```

Figure 4.3: TAILOR instruments Linux to generate the kernel wakeup event into the software channel.

create different channels for the two types of events. Figure 4.4 lists the channels TAILOR creates for the Lucene workload. There are two per-CPU software channels, `kernel_1st_channel` for syscall and pagefault events and `kernel_2nd_channel` for other OS signals, one per-task software channel for a Lucene server, one channel for the proxy, and one channel for a JVM.

Figure 4.3 shows an example of the `wakeup` event. TAILOR instruments Linux's `try_to_wake_up` function to generate the wakeup event into the current CPU's `kernel_2nd_channel`. Next, we discuss how the SHIM profiler records TALECHAIN events from software channels into the timeline stream.

4.2.2 Tailor Profiler and Proxy

Figure 4.5(a) shows the architecture of TAILOR, which consists of two parts: a single-thread SHIM profiler and a single-thread event-driven application-level network proxy. For web services that support local-node redundancy, TAILOR uses a local redundant backup server to mitigate the impact of unavoidable hazardous system behaviors. To avoid interference between TAILOR components, TAILOR binds them to different

```
1  union software_channel {
2      //The kernel events
3      struct wakeup_event wakeup;
4      struct scheduler_event scheduler;
5      struct softirq_event softirq;
6      struct tcp_event tcp;
7      ...
8      //The JVM events
9      struct JVM_event jvm;
10     ...
11     //The application events
12     struct app_events request;
13   }
14 };
15 // Per-CPU channels for high-frequency syscall and pagefault events.
16 DEFINE_PER_CPU(union software_channel *, kernel_1st_channel);
17 // Per-CPU channels for other kernel events.
18 DEFINE_PER_CPU(union software_channel *, kernel_2nd_channel);
19 // Per-Task channels for Lucene servers
20 union software_channel * lucene_server;
21 union software_channel * lucene_redundant_server;
22 // Per-JVM channels
23 union software_channels * JVM_channels;
24 // The Tailor proxy channel
25 union software_channels * proxy_channel;
```

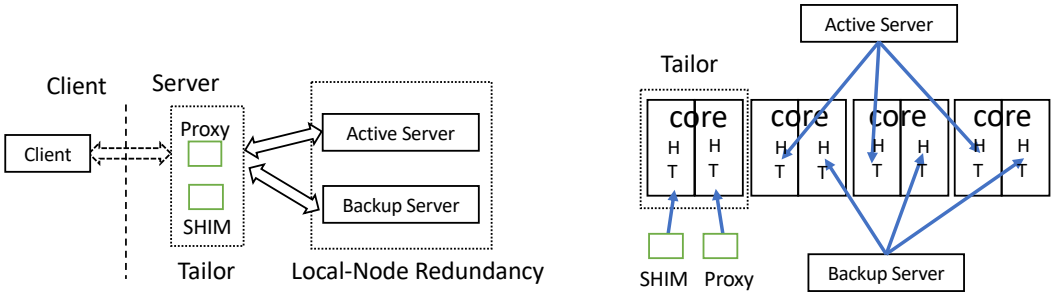
Figure 4.4: TAILOR creates software channels for the operating system, the JVMs, the proxy, and Lucene search servers.

hardware contexts with `setaffinity()`. On Intel multicore hardware with two SMT threads per core, TAILOR binds the proxy and the profiler to different SMT hardware threads of a dedicated core, then pins the active server and the backup server to different sets of SMT threads of other cores. While SHIM continuously records TALECHAIN events from software channels to the timeline stream, the proxy maintains a request table that records the information of in-flight requests and the last N finished requests such as the start and the end of the server-observable period, as shown in Figure 4.5(b).

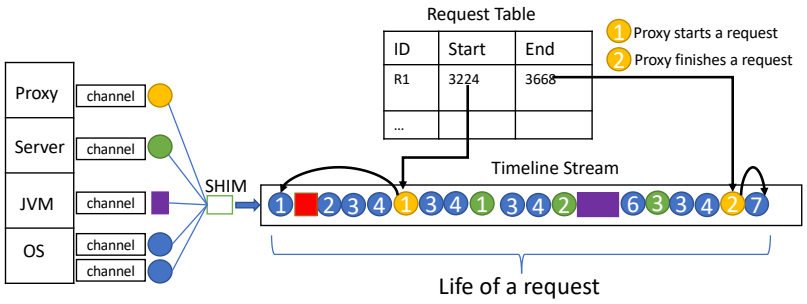
The Shim Profiler Figure 4.6 shows the pseudocode of the SHIM profiler that continuously checks whether software channels have new events, and records new events into the timeline stream. The timeline stream (line 8) is a ring buffer, in which each slot (line 2) has the recorded event and two other fields, `read_timestamp` and `channel_id`, indicating when and where the profiler reads this event (from).

The profiler detects new events by checking whether the timestamp of the current event in the channel is different from the last event, thus it uses an event cache (line 11) storing the last events from all software channels. When reading events from channels, since there is no synchronization between event generators and the profiler, also as shown in Figure 4.3, generating an event requires multiple writes which are not atomic, thus it is possible to read corrupted events.

TAILOR uses two approaches to reduce the time window in which corrupted events may be read: 1) When sampling channels, the profiler copies current events to the



(a) TAILOR binds the SHIM profiler and the proxy on the same core but on different SMT threads, and pins the active server and the backup server on different sets of SMT threads of other cores.



(b) The SHIM profiler records TALECHAIN events into a timeline stream. The proxy indexes the timeline stream with requests identifiers marking the start and the end of the server-observable period.

Figure 4.5: The architecture of TAILOR

```
1  /******** Timeline Stream *****/
2  struct timeline_stream_slot
3  {
4    u64 read_timestamp;
5    u32 channel_id;
6    union software_channel event;
7  };
8  struct timeline_stream_slot *stream;
9
10 /****** SHIM Profiler ****/
11 struct events_cache {
12   union software_channel *last;
13   union software_channel *current;
14 };
15
16 struct event_cache * caches;
17 union software_channel * source_channels;
18
19 while(true)
20 {
21   for (i=0; i<nr_channel; i++)
22   {
23     current_cache = caches[i].current;
24     last_cache = cache[i].last;
25     copy_event(source_channels[i], current_cache)
26     if (current_cache->timestamp != last_cache->timestamp)
27     {
28       new_event = current_cache;
29       copy_cache_event_to_stream(current_cache, stream);
30       swap_current_and_last_cache(caches + i);
31       if (new_event has pre-programed actions)
32         take_actions_for_event(new_event);
33     }
34   }
35  }
```

Figure 4.6: The SHIM profiler continuously records new TALECHAIN events from software channels into the timeline stream and takes preprogrammed actions for certain events.

event cache (line 25 in Figure 4.6); 2) When generating events, the generator updates the event timestamp last so that if the profiler detects a timestamp change (line 31 in Figure 4.3), it is guaranteed to see other changes on processors that support Total Store Ordering, such as x86.

After comparing timestamps, if the current event is a new one, the profiler swaps the current and the last cache event, records the new event into the timeline stream, then takes preprogrammed actions if there are any (line 26-33 in Figure 4.6).

The Tailor Proxy The TAILOR proxy is an event-driven application-level proxy. As shown in the pseudocode of Figure 4.7, the proxy listens on three types of events: incoming server requests from clients, outgoing server responses from the active and backup servers, and TALECHAIN requests from clients for analyzing the requests with slow client-observed latency. The profiler may send duplicated requests (will be discussed in the next section) to both servers at the same time, thus when forwarding server responses, the proxy checks whether the request is already finished to avoid sending duplicated responses (line 29). The checking involves looking up the request

status from the request table that keeps the information of both in-flight requests and the last N finished requests such as the processing time, the server-observable period in the timeline stream, as well as the status of the request (lines 2-9).

The proxy analyzes TALECHAIN events for slow requests and presents a timeline view showing how system behaviors affect the request latency. It detects slow requests in two places: 1) When forwarding responses, it checks whether the proxy-observed latency is longer than the target latency (line 35). 2) The proxy also accepts TALECHAIN requests from clients asking the proxy to analyze a request with a long client-observed latency (line 38). When analyzing a slow request, the proxy chains the TALECHAIN events of a request and presents a timeline graph. Figure 4.8 shows the timeline view of a normal Lucene request which finished in 450 μs. Each bar represents one TALECHAIN event in the period. The red bars mark the five stages of processing requests: the proxy receives the request, then the Lucene coordinator task receives the request, the Lucene worker starts to process the request, the Lucene worker finishes the request, and finally the proxy finishes the request. The blue arrows show the request flow of these five stages, while the green arrows show wake-up chains pointing from wakeup events to their corresponding schedule events.

4.2.3 Local-Node Redundancy

Using redundancy to mitigate the latency impact of random events is a well-known technique, but currently only used in distributed systems in which the redundancy is already provided by replicas. Dean and Barroso [2013] use hedged requests to reduce the tail-latency variability of large-scale web services, whereby clients send out the same request to multiple replica machines and use the fastest response.

We use hedged requests to reduce the impact of random unavoidable hazardous behaviors on a local machine. For web services that support distributed redundancy, it is straightforward to set up local-node redundancy. TAILOR starts another backup server on the same machine and binds it to a different set of SMT threads. Whenever the profiler detects hazardous systems events, it immediately swaps the active and the backup server, then resends all in-flight requests to the new active server. The proxy uses the first response of hedged requests and rejects later ones when forwarding responses.

The major unavoidable hazardous behaviors for the Lucene search workload are JVM pauses caused by JVM restarts and garbage collections. Figure 4.9 shows how the profiler takes real-time actions, in the event of JVM restart and garbage collections. For the JVM-restart event, after sending out hedged requests to the new active server, the profiler also sends warmup requests to the restarted server in order to avoid hitting delays on the next switch.

The key difference between our approach and the hedged requests used in distributed systems is the decision accuracy. TAILOR only sends hedged requests after detecting hazardous system behaviors, while as shown in Dean and Barroso [2013], hedged requests are unconditionally sent out after some brief delay because it is not possible to monitor fine-grain server behaviors from remote clients. If extra de-

```
/****** Tailor Proxy ****/
struct request {
  u64 start_timestamp;
  u64 finish_timestamp;
  struct timeline_stream_slot *stream_start;
  struct timeline_stream_slot *stream_end;
  u32 rid;
  u32 flag;
};

struct request *request_table;

while(true)
{
  fds = epoll_wait();
  ...
  if (client request)
  {
    rid = parse(request);
    gen_proxy_receive_signal(rid);
    request_table[rid]->start_timestamp = now;
    request_table[rid]->stream_start = stream_pos;
    ...
    forward_request();
  }
  if (server response)
  {
    rid = parse(response);
    if (request rid is already finished)
      continue;
    gen_proxy_finish_signal(rid);
    request_table[rid]->start_timestamp = now;
    request_table[rid]->stream_end = stream_pos;
    forward_response();
    if (request rid is slow)
      analyze_slow_request(request_table[rid]);
  }
  if (TaleChain request)
  {
    rid = parse(request);
    analyze_slow_request(request_table[rid]);
  }
}
```

Figure 4.7: The proxy forwards requests and responses between clients and the active and backup servers, maintains the request table, and analyzes slow requests.

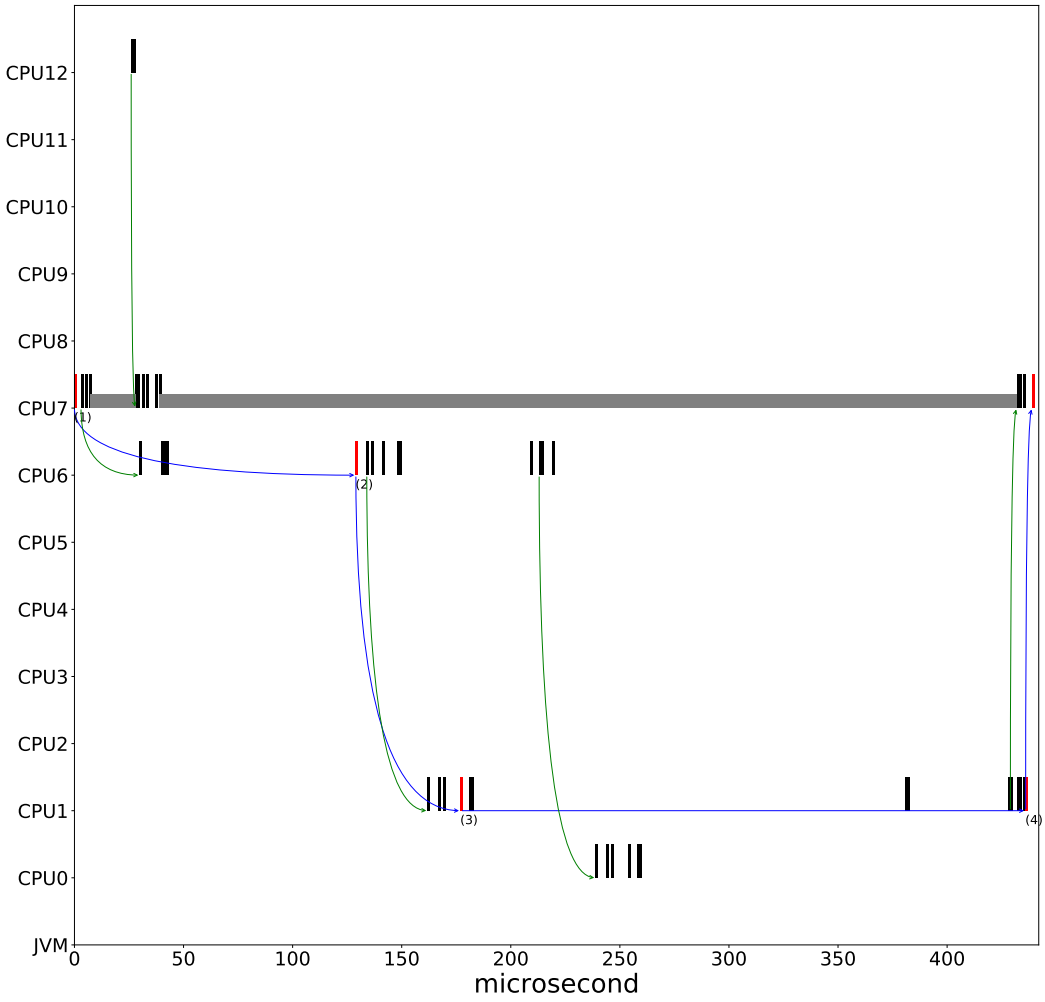


Figure 4.8: TAILOR presents a timeline view of a Lucene request. Each bar represents one TALECHAIN event. The red bars mark the five stages of processing requests: the proxy receives the request, then the Lucene coordinator task receives the request, the Lucene worker starts to process the request, the Lucene worker finishes the request, and finally the proxy finishes the request. The blue arrows show the request flow of these five stages, while the green arrows show wake-up chains pointing from wakeup events to their corresponding schedule events.

```
while(true)
{
  for (i=0; i<nr_channel; i++)
  {
    current_cache = caches[i].current;
    last_cache = cache[i].last;
    copy_event(source_channels[i], current_cache)
    if (current_cache->timestamp != last_cache->timestamp)
    {
      new_event = current_cache;
      copy_cache_event_to_stream(current_cache, stream);
      swap_current_and_last_cache(caches + i);
      if (new_event is JVM_restart or JVM_garbage_collection)
      {
          swap_active_backup_server();
          for (r in flight requests)
            send_request_to_the_active_server(r)
      }
    }
  }
 }
```

Figure 4.9: The Shim profiler mitigates the impact of JVM pauses by sending hedged requests after detecting hazardous events.

lays are acceptable, instead of directing delayed requests to a backup server on the local machine, Tailor can also dispatch the requests to backup servers on remote machines.

4.3 System Setup

Platform We use a 2.1 GHz Intel Xeon-D 1541 Broadwell [Intel, 2015] processor with eight two-way SMT cores, a 12 MB shared L3. Each core has a private 256 KB L2, a 32 KB L1D and a 32 KB L1I. The TurboBoost maximum frequency is 2.7 GHz, but we disable it to control the variability. TDP is 45 W. The machine has 32 GB of memory and two Gigabit Ethernet ports. We set the package-level deep-sleep state to C6.

We use Ubuntu 16.04, Linux version 4.18.0. We expose a memory buffer to user space hosting operating system software channels. User-space channels are exposed as shared files that can be mapped to memory.

We execute the Lucene search server in the OpenJDK9 JVM (build jdk-9+181). The JVM runs in server mode with a 2 GB heap, 7 parallel GC threads, the G1 garbage collector, and the C1 compiler. We use the Lucene search server from the Lucene performance regression framework, which uses the Lucene library (LUCENE-8324). The Lucene search server has one coordinator and six workers bound to CPU cores 0-6. When we use local-node redundancy, we create another Lucene search server and bind two servers on the same set of cores but different SMT threads. We implement the Tailor proxy and the Shim profiler in C, and bind them to core 7 but on different SMT threads.

The Search Workload We use the Lucene performance regression framework to build indexes of the first 10 K files from Wikipedia's English XML, then create 1000 search queries that have the server-execution time less than 2.5 ms as shown in Figure 4.10(a). We send the search requests from another machine that has the same specifications as the server. The two machines are connected via an isolated Gigabit switch. For each experiment, we perform 2 invocations, one warmup iteration and another experimental iteration. For each invocation, the client loads 1000 requests and sends requests 20 times (iterations). The client issues search requests at random intervals following an exponential distribution around the 1000 queries per second (QPS) mean rate. We report the tail latency of the 20 iterations of the second invocation. Figure 4.10(b) shows both the server-observed and the client-observed 99%ile latency and the maximum tail latency (the slowest request) with one Lucene search server under the 1000 QPS light load (no TAILOR and the backup server). The client-observed maximum tail latency is consistently much higher than the server-observed latency, 40-46 ms verses 2-36 ms.

Summarizing, our queries are chosen for a maximum server time of 2.5 ms, but we see server-observed tail latencies of 2-36 ms and client-observed tail latencies of 40-46 ms. The next section shows how TAILOR identifies hazardous system behaviors and reduces the client-observed maximum latency to 3-5 ms.

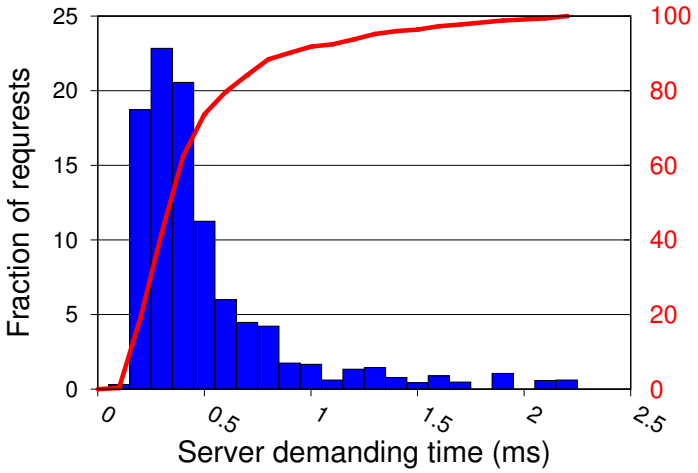
4.4 Hazardous System Behaviors and Optimizations

This section describes four major hazardous system behaviors identified by TAILOR and their solutions. For each behavior, we present a TALECHAIN timeline view showing how the behavior affects request latency and a tail-latency result, demonstrating the progress of reducing the impact of the hazardous behaviors step by step.

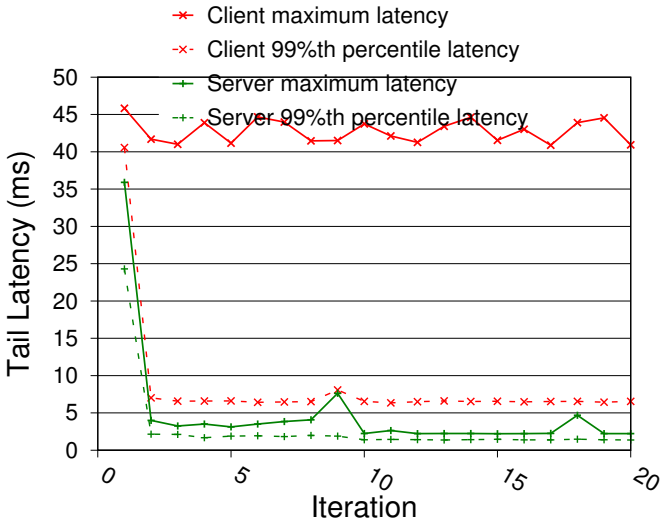
TCP Buffering The maximum client-observed tail latency for our workload is consistently above 40 ms across all 20 iterations (Section 4.3). We run this workload with TAILOR, which detects the request and presents a TALECHAIN timeline view shown in Figure 4.11.

The request illustrated in Figure 4.11 quickly finished the first four steps in less than 1 ms. However, after the Lucene server wrote the query response to the socket, a 40 ms delay occurred before the network stack transmitted the socket buffer to the ethernet card (the purple arrow). There are two TCP-xmit events in the period trying to transmit the data, but the first try failed. The status field of the first TCP-xmit event explains the reasons: the operating system was waiting for a TCP acknowledgment from the client before transmitting the response.

This hazardous behavior is a combination of The TCP Nagle algorithm and delayed TCP ACKs. On our experimental machines, by default, the TCP socket uses the Nagle algorithm to automatically concatenate small TCP segments in order to avoid sending out a large number of small packets. It delays sending out new TCP segments until receiving the acknowledgment for the previous ones. However, TCP



(a) The server-execution time distribution of 1000 search requests.



(b) Client and server tail latencies

Figure 4.10: The tail latency of sending 1000 requests in 20 iterations at 1000 QPS.

does not send an ACK immediately after receiving a segment, but waits for a period which can be up to 500 ms, expecting to piggyback the ACK on the next out-going segment.

In this case, the Lucene server was waiting for the client's ACK, but the client delayed sending the ACK since it was waiting for the next out-going data. Unfortunately, this request was the last one in the iteration, thus there was no out-going data for the client to piggyback on. After the timeout, the client sent out the ACK, then the server transmitted the response immediately, which is also why the client-observed tail latency of the request is much longer than the server-observed one.

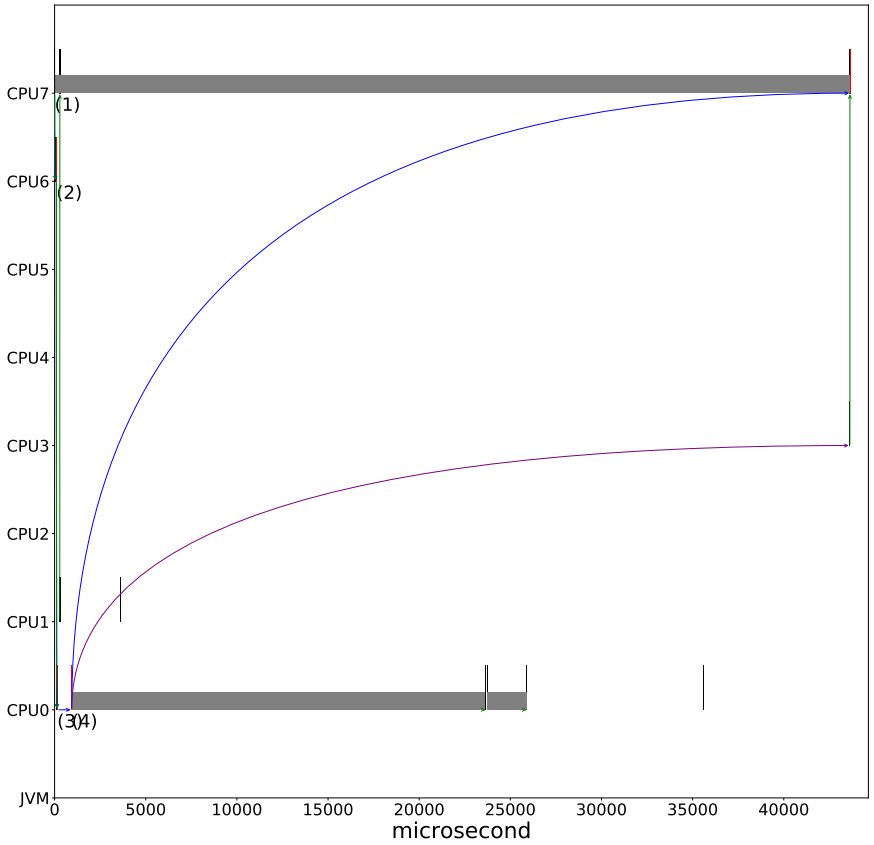
After identifying this hazardous behavior, we add a `TCP-NODELAY` flag when creating the TCP socket to disable the Nagle algorithm. Figure 4.11(b) shows that this simple action reduces the client-side tail latency from 40-46 ms to 3-40 ms.

Slow Page Faults After The Deep Sleep Figure 4.12(a) presents another hard-to-find hazardous system behavior: long page faults taking about 5 ms (purple bars). The page faults happened while the Lucene workers processed the search requests.

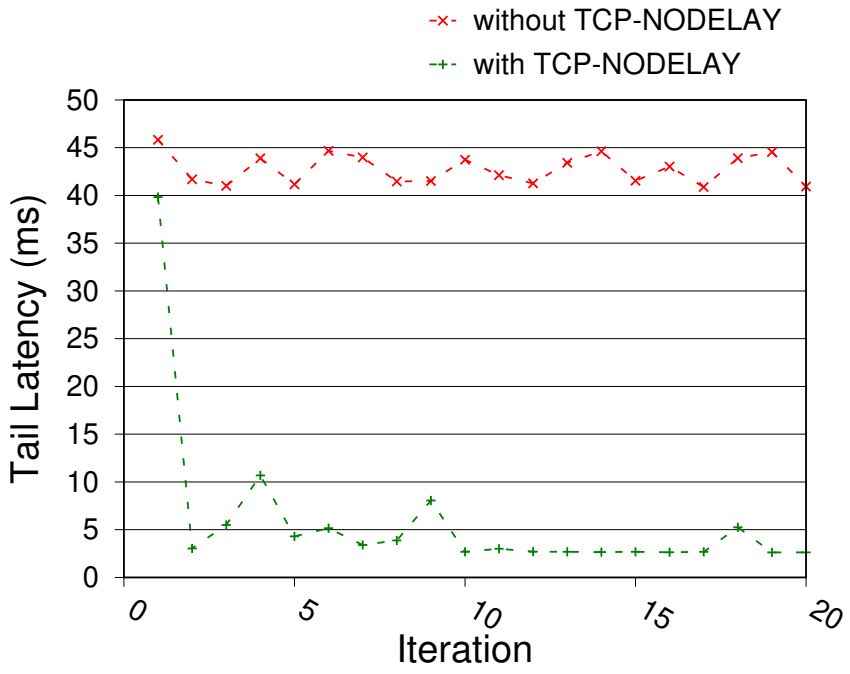
We analyzed TaleChain events preceding the request period, and also used the ftrace profiler to get the traces of Linux's page fault handler `handle_mm_fault()` finding that: 1) The long page faults spend most of the time in the function `do_huge_pmd_anonymous_page()` on zeroing newly allocated huge pages. 2) Long page faults only happen when more than three page faults are triggered at the same time on three different Lucene workers after a long idle period.

The information suggests that the root cause of slow page faults is extremely poor memory performance after a deep CPU sleep. Linux puts idle CPU cores into a sleep state. The longer the idle period is, the deeper sleep state the CPU enters. We enable the package-level C6 deep-sleep state on our machines, in which the CPU saves the architectural state to memory, flushes caches, and turns off on-chip components. Schöne et al. [2015] show that it takes modern Intel processors about 0.3 ms to leave the C6 deep-sleep state. Our result indicates that the latency of exiting from the deep sleep contributes little to the long page faults, but the majority of the overhead is from poor performance of memory operations after exiting from the deep-sleep state. The slow memory operations can be caused by strong contention between cores when multiple Lucene workers zero huge pages concurrently.

Instead of disabling the deep-sleep state which would increase energy consumption, we address the slow page faults by turning on the `AlwaysPreTouch` JVM option. When the option is on, the HotSpot JVM pre-touches the Java heap during JVM initialization so that subsequent allocations from the Java heap do not trigger page faults. As shown in Figure 4.12(b), the second-highest client-side tail latency in the 4th iteration is reduced from 10 ms to 2.5 ms. Notice that after 10 iterations, turning on the option does not reduce the tail latency further. This is because we run 20 iterations in the same JVM, thus after many iterations, the Java heap may have been touched already. Note that the tail latency of the first iteration is still 40 ms, after addressing the first two hazardous behaviors.

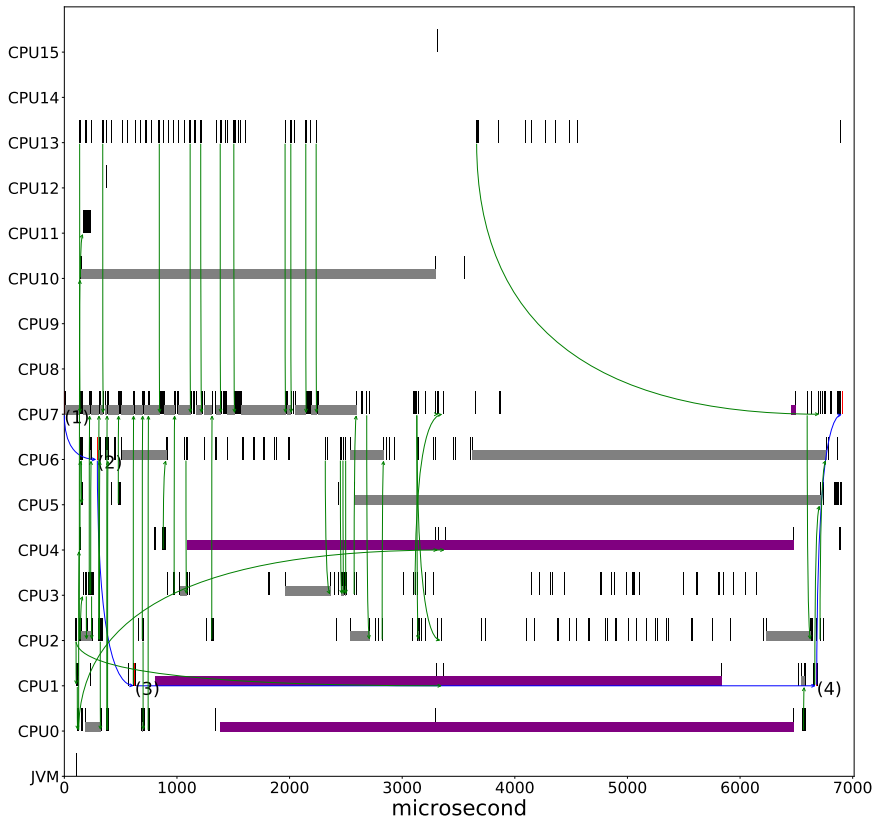


(a) The TaleChain timeline view of the slow request affected by the TCP Nagle algorithm. The green arrows show wake-up chains, the blue arrows represent the flow of the search request, and the purple arrow shows the TCP buffering caused by the TCP Nagle algorithm.

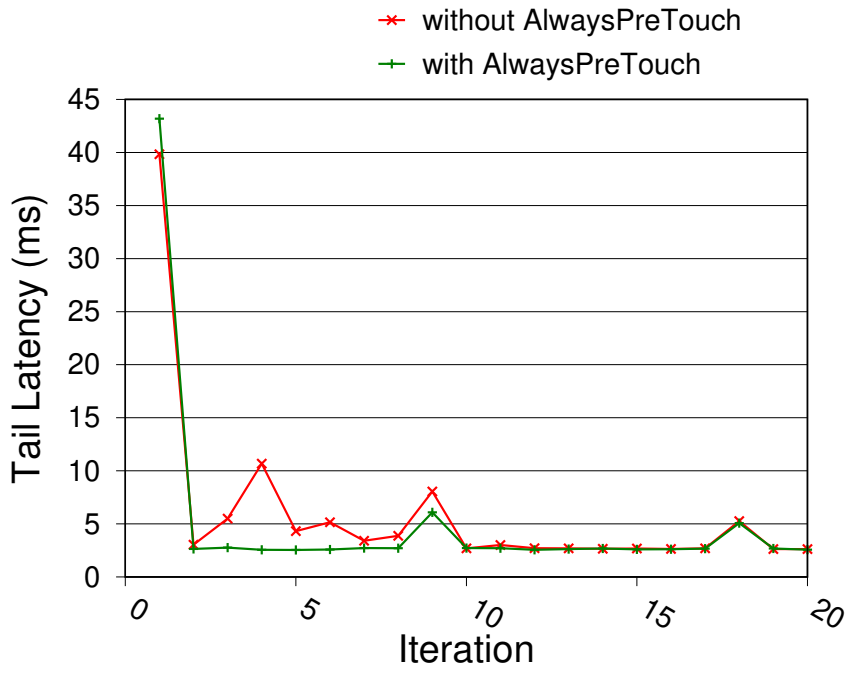


(b) Disabling the TCP Nagle algorithm reduces the maximum client tail latency.

Figure 4.11: The operating system buffers the request response. Disabling the TCP Nagle algorithm avoids the hazardous behavior.



(a) The TaleChain timeline view of slow requests affected by long page faults (purple bars).



(b) Maximum client tail latency.

Figure 4.12: Extremely long page faults after the deep sleep slow down requests. Pre-touching the JVM heap avoids triggering the hazardous behavior, reducing the client-side tail latencies of iterations 3-8 under 3 ms consistently. Notice that other peaks caused by JVM pauses have not been addressed yet.

JVM Pauses The long tail of the first iteration is not caused by loading the Lucene index data from the storage system into memory, because as discussed in the last section, we execute one warmup invocation with 20 iterations before the experiment to load the index data into memory.

The root cause actually is a large number of short pauses caused by loading Java classes when the JVM executes the first few requests after a restart. Loading and compiling Java classes is a complex and non-scalable process, involving many JVM components. Figure 4.13(a) shows this chaotic process. Each green arrow represents one wake-up chain pointing from the wakeup event to the corresponding schedule event. Most of the wake-up chains in the graph are caused by the lock-related interaction between JVM threads. The figure shows that Lucene workers (CPU cores 0-7) continuously wake up each other, indicating complex interactions and strong lock contention between them.

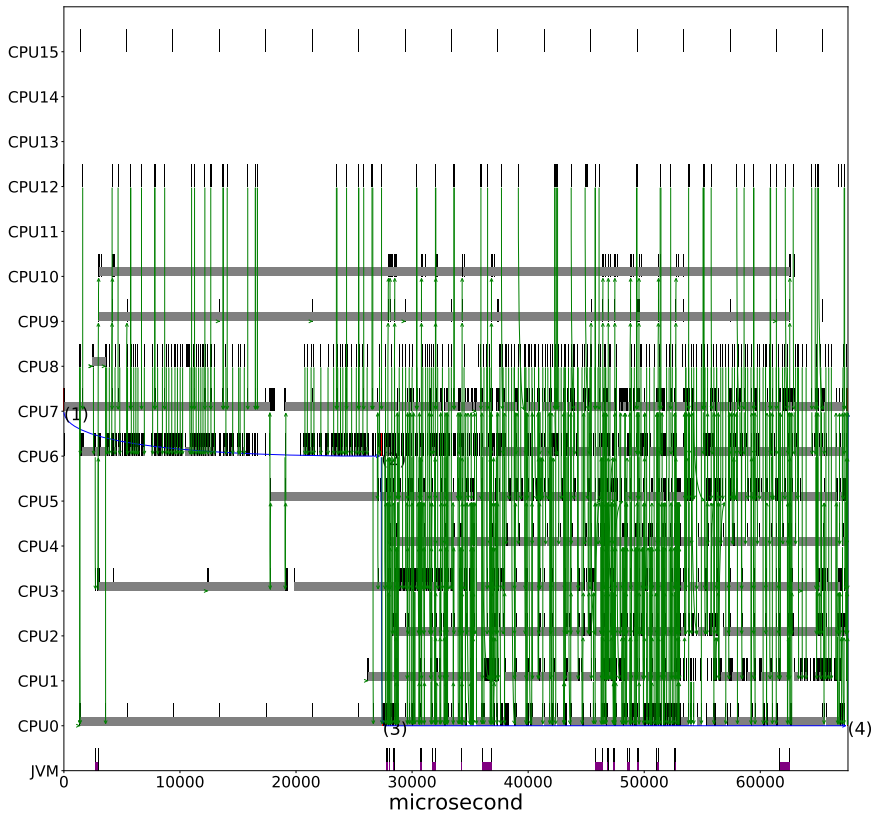
The other two peaks of the client tail latency (the red line in Figure 4.14) are caused by another type of JVM pauses, the HotSpot JVM's nursery garbage collection that stops all JVM threads when performing the collection. Figure 4.13(b) shows a timeline view of the request paused by the collection (the purple bar).

Tailor reduces the impact of unavoidable JVM pauses by using local-node redundancy. It creates a backup Lucene search server and binds it to a different set of SMT threads. Whenever Tailor's profiler detects JVM-restart events and garbage collection events, it switches the active server and the backup server, then re-sends all in-flight requests to the new active server. Figure 4.14 shows that the approach successfully hides the impact of garbage collection, reducing the second and the third peaks of client-side tail latency from 7 ms to under 3 ms, and reduces the impact of class loading significantly, decreasing the tail latency of the first iteration from 41 ms to 4 ms. The reason why the tail latency of the first iteration is still 1 ms longer than other iterations after addressing the JVM pauses is that after Tailor switches the active and the backup server, Tailor immediately sends out warmup requests to the restarted server. As shown in Figure 4.13(a), class loading is a long and resource consuming process. Because the active and the backup server share core resources, the class loading caused by the warmup requests on the restarted server may affect the performance of the active server. The next chapter discusses a mechanism to control such interference between same-core SMT threads.

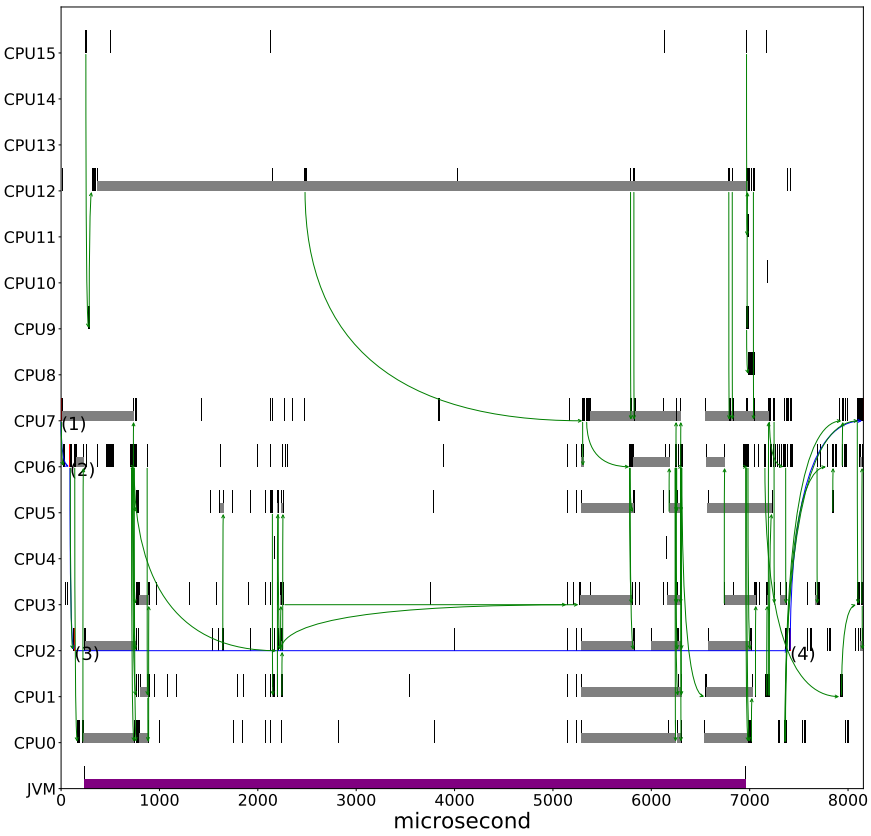
After addressing the four hazardous system behaviors, the client-side maximum tail latency is reduced nine-fold, from 41-46 ms to 3-5 ms.

4.5 Related Work

Unlike general-purpose profilers, Tailor focuses on profiling, analyzing, and acting on long latency requests. The most related work on diagnosing and fixing tail latency sources in application and system software [Li et al., 2014; Zhang et al., 2016; Leverich and Kozyrakis, 2014] and hardware and its configuration [Li et al., 2015] offer deep insights to causes and one-time fixes, but do not propose a tool that systematically



(a) The TaleChain timeline view of the slow request delayed by the class loading. A massive number of wake-up events (the green arrows) show the chaotic interactions between Lucene workers (CPU cores 0-7).



(b) The TaleChain timeline view of the slow request delayed by the JVM garbage collection (the purple bar).

Figure 4.13: JVM pauses caused by class loading and garbage collection affect request latency.

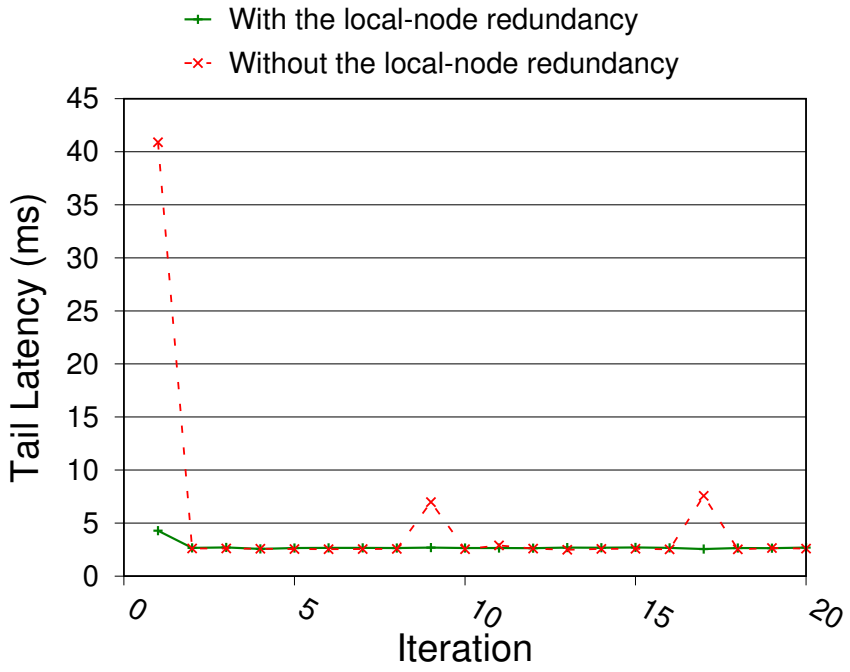


Figure 4.14: TAILOR reduces the maximum client-side tail latency by mitigating the impact of JVM pauses with the local-node redundancy.

exposes problems and counter acts unavoidable latencies as they emerge in real time.

Our work is largely orthogonal to work that improves application scheduling of multiple requests [Kannan et al., 2019] and network packets [Kaffes et al., 2019] by prioritizing requests that are delayed, regardless of the reason, to manage latency and throughput. Work that detects long latency requests and then accelerates them using DVFS, heterogeneous hardware, or parallelism are most effective when the source of latency is request work, rather than hardware or software unresponsiveness [Haque et al., 2017, 2015; Hsu et al., 2015; Lo et al., 2015; Ren et al., 2013; Kasture et al., 2015]. Our approach targets this later problem — latency due to factors other than request work.

Gray introduced software redundancy as an effective approach to building software fault-tolerant computer systems [Gray, 1985]. This redundancy can be deployed both on the same or different nodes. The Tandem NonStop System deployed a pair of processes on the same machine to tolerate software faults [Bartlett, 1981]. If the active process failed to respond a request before the deadline, the waiting request was re-sent to the backup process. Today, latency-critical large-scale cloud and other services routinely deploy replicas across multiple machines for fault tolerance and scaling. Furthermore, they use the replicas to tolerate random slow requests: a mid-tier monitors the latency of running requests and duplicates slow requests to another replica [Dean and Barroso, 2013; Kaler et al., 2017].

Today, service deployment frameworks, such as Kubernets, provide abstractions

for deploying and scaling replicas, blur the boundary between the same-machine redundancy and multi-machine redundancy, so services could adopt both redundancy approaches [What is Kubernetes?, 2019]. Tailor exploits local redundancy to handle unavoidable sources of latency, which is complementary to the multi-server latency-tolerance approach and eliminates the need for a duplicated request at the mid-tier, reducing service resource requirements.

4.6 Summary

This chapter introduces Tailor, a real-time latency controller for latency-critical web services, in which a Shim-based high-frequency profiler and an application-level network proxy are combined to continuously monitor and act on hazardous system behaviors. We show that Tailor identifies such behaviors by continuously recording and analyzing signals of system components, and uses local-node redundancy to mitigate the impact of unavoidable random hazardous behaviors. We present a case study demonstrating the diagnostic power of Tailor and the effectiveness of Tailor's real-time control actions. After adjusting system configurations to avoid operating system hazardous behaviors and mitigating the impact of unavoidable JVM pauses, Tailor reduces the tail latency of the latency-critical workload by nine-fold from 46 ms to 5 ms.

This chapter shows that Shim's high-frequency profiling enables new ways to analyze and control the fine-grain behavior of complex computer systems in real time, which can be used for optimizing tail latency of latency-critical web services. The next chapter demonstrates that fine-grain control of system components leads to a new class of online profile-guided optimizations that improves CPU utilization of the servers that run latency-critical web services. It introduces Elfen, a Shim-based job scheduler that borrows idle cycles from underutilized CPU cores of latency-critical workloads for batch workloads.

Chapter 5

Elfen

Latency-critical web applications from search to games to stock trading interact with humans or other latency-sensitive services. To retain users, they are carefully engineered to meet latency requirements and impose strict Service Level Objectives (SLOs) on tail latency. Meeting these objectives is challenging. Consequently, many servers run at low utilization (10 to 45%), which indicates a promising feedback-directed optimization opportunity: filling wasted idle periods with useful batch jobs can significantly increase server utilization without interfering with latency-critical requests.

We introduce Elfen, a job scheduler that co-runs batch jobs with latency-critical web services on the same core but on different simultaneous multithreading (SMT) contexts. Elfen instruments batch threads to monitor the status of latency-critical requests with Shim. Once batch threads detect latency-critical requests, they step out of the way and promptly return the computation resources. The scheduler is configurable such that it can also use an interference budget, stepping out of the way after some periods of time.

This chapter is structured as follows. Section 5.2 motivates Elfen using workload characteristics of Lucene. Section 5.3 describes the design and implementation of Elfen. Section 5.4 shows our evaluation methodology, and Section 5.5 evaluates Elfen.

The work described in this chapter is published in "Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading" [Yang et al., 2016a].

5.1 Introduction

Latency-critical web services, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a *tail constraint*, such as that 99% of requests must complete within 100 ms [Dean and Barroso, 2013; DeCandia et al., 2007; He et al., 2012; Yi et al., 2008]. Many such services, such as Google Search and Twitter [Kanev et al., 2015; Dean and Barroso, 2013; Delimitrou and Kozyrakis,

2014], systematically underutilize the available hardware to meet SLOs. Furthermore, servers often execute only one service to ensure that latency-critical requests are free from interference. The result is that server utilizations are as low as 10 to 45%. Since these services are widely deployed in large numbers of datacenters, their poor utilization incurs enormous commensurate capital and operating costs. Even small improvements substantially improve profitability.

Meeting SLOs in these highly engineered systems is challenging because: (1) requests often have variable computational demands and (2) load is unpredictable and bursty. Since computation demands of requests may differ by factors of ten or more and load bursts induce queuing delay, overloading a server results in highly non-linear increases in tail-latency. The conservative solution providers often take is to significantly over-provision.

Interference arises in chip multiprocessors (CMPs) and in simultaneous multithreading (SMT) cores when contending for shared resources. A spate of recent research explores how to predict and model interference between different workloads executing on distinct CMP cores [Novaković et al., 2013; Mars et al., 2011; Lo et al., 2015; Delimitrou and Kozyrakis, 2014], but these approaches target and exploit large scale diurnal patterns of utilization, e.g., co-locating batch workloads at night when load is low. Lo et al. [2015] explicitly rule out SMT because of the highly unpredictable and non-linear impact on tail latency (which we confirm) and the inadequacy of high-latency OS scheduling. Zhang et al. [2014] do not attempt to reduce SMT-induced overheads, but rather they accommodate them using a model of interference for co-running workloads. Their approach requires ahead-of-time profiling of all co-located workloads and over-provisioning. Prior work lacks dynamic mechanisms to monitor and control fine-grain behavior of batch workloads on SMT with low latency.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the Elfen[1] scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granularities is inadequate, motivating fine-grain mechanisms.

Elfen introduces mechanisms to control thread execution and a runtime that monitors and schedules threads on distinct hardware contexts (*lanes*) of an SMT core. Elfen pins latency-critical requests to one SMT lane and batch threads to $N-1$ partner SMT lanes on a N-way SMT core. A *batch* thread monitors a paired lane reserved for executing latency-critical requests. (We use '*requests*' for concision.) The simplest *borrow idle* policy in Elfen ensures mutual exclusion—requests execute without interference. Batch threads monitor the request lane and when the request lane is occupied, they release their resources. When the request lane is idle, batch threads execute. We introduce `nanonap`, a new system call, that disables preemption and invokes `mwait` to

[1] In the Grimm fairy tale, *Die Wichtelmänner*, elves borrow a cobbler's tools while he sleeps, making him beautiful shoes.

release hardware resources quickly—within ~3 000 cycles. This mechanism provides semantics that neither yielding, busy-waiting, nor `futex` offer. After calling `nanonap`, the batch thread stays in this new kernel state without consuming microarchitecture resources until the next interrupt arrives or the request lane becomes idle. These semantics ensure that requests incur no interference from batch threads and pose no new security risks. Since the batch thread is never out of the control of the OS, the OS may preempt it as needed. The shared system state that Elfen exploits is already available to applications on the same core, and Elfen reveals no additional information about co-runners to each other.

We inject scheduling and Shim profiling mechanisms into batch applications at compile-time. A binary re-writer could also implement this functionality. The instrumentation frequently checks for a request running on the paired SMT lane by examining a shared memory location. When the batch thread observes a request, it immediately invokes `nanonap` to release hardware resources. This policy ensures that the core is always busy, but it only utilizes one SMT lane on a two-way SMT core at a time.

More aggressive borrowing policies use both lanes at once by giving batch threads a budget that limits overheads imposed on requests, ensuring that SLOs are met. The budget is shaped by the SLO, the batch workload's impact on the latency-critical workload, and the length of the request queue. These policies monitor the request in various ways, via high-frequency Shim profiling [Yang et al., 2015a].

We implement Elfen in the Linux kernel and in compile-time instrumentation that self-schedules batch workloads, using both C and Java applications, demonstrating generality. For our latency-sensitive workload, we use the widely deployed Apache Lucene open-source search engine [Apache Lucene, 2014]. Prior work shows Lucene can be configured to have performance characteristics and request demand distributions similar to the Bing search engine [Haque et al., 2015]. We evaluate Elfen co-executing a range of large complex batch workloads on two-way SMT hardware. On one core, Elfen's borrow idle policy achieves peak utilization with essentially no impact on Lucene's 99th percentile latency SLO. Elfen improves core utilization by 90% at low load and 25% at high loads compared to a core dedicated only to Lucene requests. It consistently achieves close to 100% core utilization, the peak for this policy — one of the two hardware contexts always busy. On an eight core CMP, the borrow idle policy usually has no impact or slightly improves 99th percentile latency because cores never go to sleep. Occasional high overheads at high load may require additional interference detecting techniques. Improvements in CMP utilization are more substantial than for one core because at low load, many cores may be idle. Elfen consistently achieves close to 100% of the no-SMT peak, which is also the borrow idle policy's peak utilization.

Choosing a policy depends on provider workloads, capacity, and server economics, including penalties for missed SLOs and costs for provisioning servers. Providers currently provide excess capacity for load bursts and SLOs slack for each request. Our approach handles both. Our most conservative borrow idle policy steps out of the way during load bursts and suits a setting where the penalties for missed SLOs

are very high. Our more agressive policies can soak up slack and handle load bursts. They offer as much as two times better utilization but at the cost of higher median latencies and higher probability of SLO violations. For server providers with latency-critical and batch workloads, the main benefit of our work is to substantially reduce the required number of servers for batch workloads.

In summary, contributions of this chapter include:

- analysis of why latency-critical workloads systematically underutilize hardware and the opportunities afforded by idle periods;
- `nanonap`, a system call for fine-grain thread control;
- Elfen, a scheduler that borrows idle cycles from underutilized SMT cores for batch workloads without interfering with latency-critical requests;
- a range of scheduling policies;
- an evaluation that shows Elfen can substantially increase processor utilization by co-executing batch threads, yet still meet request SLOs; and
- an open-source implementation on github [Yang et al., 2016b].

Our approach requires only a modest change to the kernel and no changes to application source code, making it easy to adopt in diverse systems and workloads.

5.2 Background and Motivation

We motivate our work with workload characteristics of latency-critical services; the non-linear effects on latency from uncontrolled interference with SMT; the opportunity to improve utilization availed by idle resources; and requirements on responsiveness dictated by idle periods. Section 5.4 describes our evaluation methodologies.

Processing demand The popular industrial-strength Apache Lucene enterprise search engine is our representative service [Apache Lucene, 2014]. Prior work shows that services such as Bing search, Google search, financial transactions, and personal assistants have similar computational characteristics [Dean and Barroso, 2013; Delimitrou and Kozyrakis, 2014; Ren et al., 2013; Haque et al., 2015; Hauswald et al., 2015]. Figure 5.1 plots the distribution of request processing times for Lucene executing in isolation on a single hardware context. The bars (left y-axis) show that most requests are short, but a few are very long. This high variance of one to two orders of magnitude is common in such systems.

Load sensitivity This experiment shows that high load induces non-linear increases on latency. We assume a 100 ms service level objective (SLO) on 99th percentile latency for requests. A front end on separate hardware issues search requests at random intervals following an exponential distribution around the prescribed requests per second (RPS) mean rate. As soon as Lucene completes a request, it processes the next request in the queue. If no requests are pending, it idles. We show results for a single Lucene worker thread running on one core.

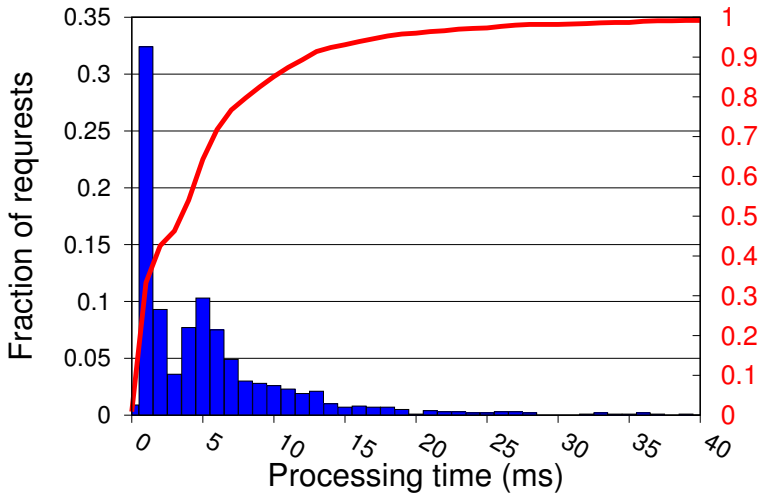


Figure 5.1: Highly variable demand is typical for latency-critical workloads. Lucene demand distribution with request processing time on x-axis in 1 ms buckets, fraction of total on left y-axis, and cumulative distribution red line on right y-axis.

Figure 5.2 shows Lucene percentile latencies and utilization as a function of RPS only on one lane of a two-way SMT core using one Lucene task. The two graphs share the same x-axis. The top graph shows median, 95th, and 99th percentile latency for requests, the bottom graph shows CPU utilization which is the sum of the fraction of time the lanes are busy normalized to the theoretical peak for a system with SMT disabled. The maximum utilization is 2.0, but the utilization in Figure 5.2 never exceeds 1.0 because only one thread handles requests, so only one lane is used. As RPS increases, median, 95th, and 99th percentile latencies first climb slowly and then quickly accelerate. The 99th percentile hits a wall when RPS rise above 120 RPS, while the request lane utilization is only 70% at 120 RPS, leaving the two-way SMT core substantially underutilized when operating at a *practical* load for a 100 ms 99th percentile tail latency target.

Random request arrival times and the high variability of processing times combine to produce high variability in queuing times and non-linear increases in latencies at high RPS. As we show next, adding a co-runner on the same core using SMT has the effect of throttling the latency-critical workload, effectively moving to the right in Figure 5.2. Movements to the right lead to increasingly unpredictable latencies, and likely violations of the SLO.

Simultaneous Multithreading (SMT) This section gives SMT background and shows that simultaneously executing requests on one lane of a 2-way SMT core and a batch thread on the other lane degrades request latencies. This result confirms prior work [Zhang et al., 2014; Delimitrou and Kozyrakis, 2014; Lo et al., 2015] and

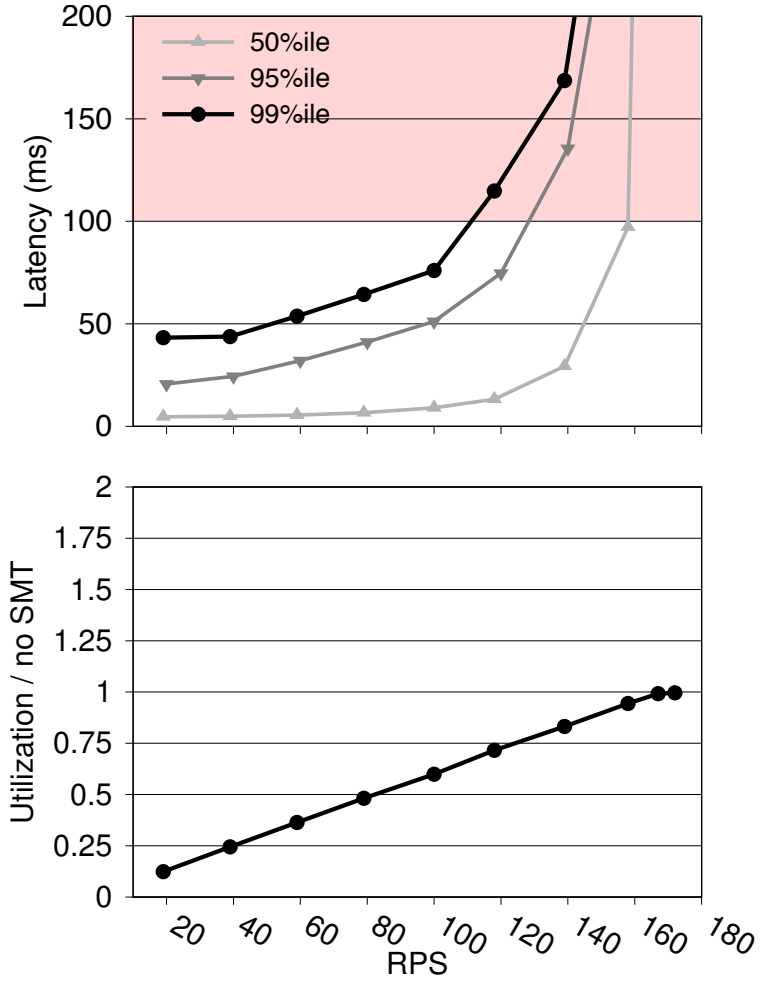
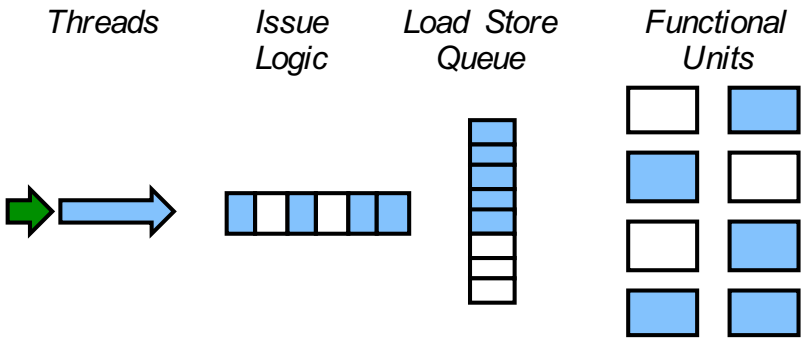
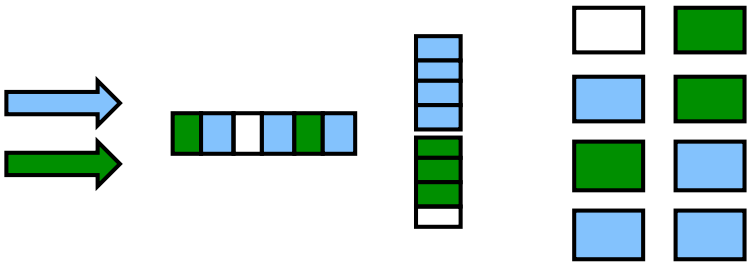


Figure 5.2: Overloading causes non-linear increases in latency. Lucene percentile latencies and utilization on one core. Highly variable demand induces queuing delay, which results in non-linear increases in latency.



(a) **Core without SMT**



(b) **Core with SMT enabled**

Figure 5.3: Simultaneous Multithreading (SMT) A single thread often underutilizes core resources. SMT dynamically shares the resources among threads.

explains why service providers often disable SMT. We measure core idle cycles to show that the opportunity for improvement is large, if the system can exploit short idle periods.

We illustrate the design and motivation of SMT in Figure 5.3. Figure 5.3(a) shows that when only one thread executes on a core at a time, hardware resources such as the issue queue and functional units are underutilized (white). Figure 5.3(b) shows two threads sharing an SMT-enabled core. The hardware implements different sharing policies for various resources. For example, instruction issue may be performed round-robin unless one thread is unable to issue, and the load-store queue partitioned in half, statically, while other functional units are shared fully dynamically. It is important to note that such policies mean that a co-running thread consumes considerable core resources *even when that thread has low IPC.*

To measure lower bounds on SMT interference, we consider two microbenchmarks as batch workloads executing on an Intel 2-way SMT core. The first uses a non-temporal store and memory fence to continuously block on memory, giving an IPC of 0.01. For instance, the Intel PAUSE instruction has a similar IPC. The other performs a tight loop doing nothing (IPC=1) when running alone. Neither consume cache or memory resources. Figure 5.4 shows the impact of co-running batch workloads on the 99% percentile latency of requests and lane utilization. Utilization improves over no co-runner significantly since the batch thread keeps the batch lane busy, but

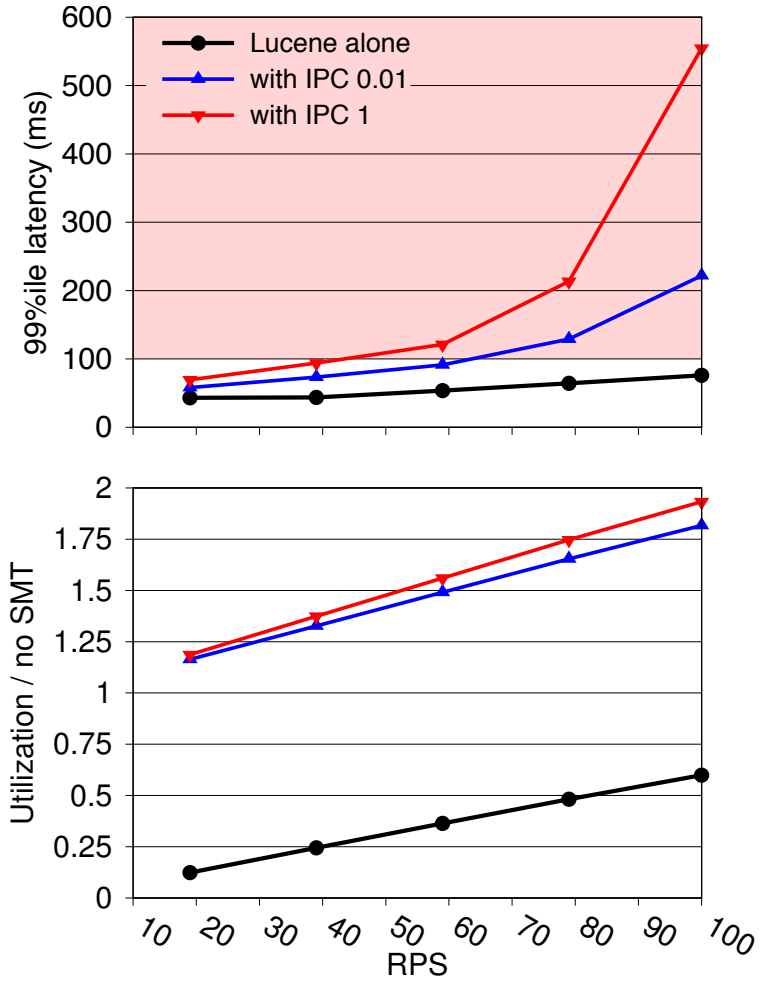


Figure 5.4: Unfettered SMT sharing substantially degrades tail latency. Lucene 99th percentile latency and lane utilization with IPC 1 and IPC 0.01 batch workloads.

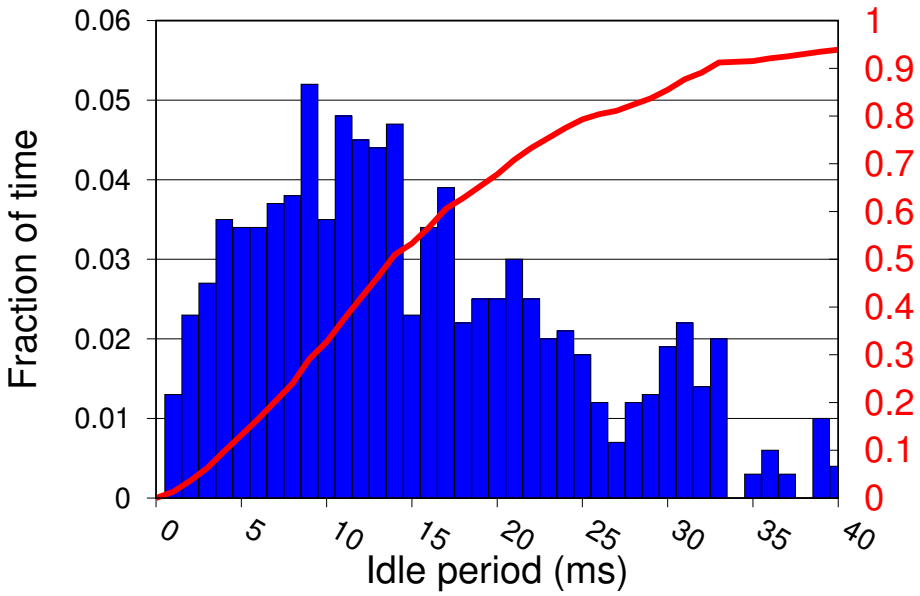


Figure 5.5: Lucene Inter-request idle times are highly variable and are frequently short. Histogram (left y-axis) shows the distribution of request idle times. The line (right y-axis) shows the cumulative time between completing one request and arrival of the next request at 0.71 lane utilization on one core at RPS = 120.

request latency degrades substantially, even when the batch thread has very low resource requirements (IPC = 0.01). For instance, at 100 RPS without a co-runner, 99th percentile latency is 76 ms. RPS must fall to around 40 RPS to meet the same 99th percentile latency with a low IPC co-runner.

Co-running moves latencies to the right on RPS curves, into the steep exponential, with devastating effect on SLOs. Because SMT hardware shares resources such as issue logic, the load store queue (LSQ), and store buffers, tail latency suffers even when the batch workload has an IPC as low as 0.01 causing the latency-critical service to violate SLOs even when the utilization of the latency-critical lane is as low as 30%. If a request is short, a co-runner may substantially slow it down without breaching SLOs. Unfortunately request demand is not known a priori. Moreover, request demand is hard to predict [Lorch and Smith, 2001; Hsu et al., 2015; Jalaparti et al., 2013; Kim et al., 2015] and the prediction is never free or perfect, thus we do not consider request prediction further.

Utilization cannot be increased by using multiple SMT lanes simultaneously without degrading latency-critical requests. However existing mechanisms can not deliver such semantic. For example, giving real-time scheduling priority to threads running a latency-critical service does not prevent multiple SMT lanes of same cores being used simultaneously since batch jobs can be scheduled on the paired SMT lanes. The strategies we explore are thus (1) to enforce mutual exclusion, executing a batch

thread only when the partner lane is idle (*borrow idle*), and (2) to give the batch thread a budget for how much it may overlap execution with requests. These strategies require observing requests, detecting idle periods, and controlling batch threads.

Idle cycle opportunities Now we explore the frequency and length of idle periods to understand the requirements on the granularity of observing requests and controlling batch threads. Figure 5.5 shows the fraction of all idle time (y-axis) due to periods of a given length (x-axis). The histogram (blue) indicates the fraction of all idle time due to idle times of a specific period, while the line (red) shows a cumulative distribution function. For example, this shows that 2.3% of idle time is contributed by idle times of 15 ms in length (blue), and 53% of total idle time is due to idle times of 15 ms or less (red). **Highly variable and short idle periods dictate low-latency observation and control mechanisms.**

5.3 Elfen Design and Implementation

This section describes the design and implementation of Elfen, in which latency-critical requests and batch threads execute in distinct SMT hardware contexts (*lanes*) on the same core to improve server utilization. Given an N-way SMT, $N-1$ SMT lanes execute batch threads, *batch lanes*, and one SMT lane executes latency-critical requests, the *request lane*. We restrict our exposition below to 2-way SMT for simplicity and because our evaluation Intel hardware is 2-way SMT.

As Figure 5.4 shows, unfettered interference on SMT hardware quickly leads to SLO violations. Elfen controls batch threads to limit their impact on tail latency. We explore borrowing policies applying the principle of either eliminating interference or limiting it based on some budget. The simplest policy enforces mutual exclusion by forcing batch threads to relinquish their lane resources whenever the request lane is executing a request. More aggressive borrowing policies add overlaping the execution of batch threads and requests, governed by a budget.

The Elfen design uses two key ideas: (1) high-frequency, low-overhead monitoring to identify opportunities, and (2) low-latency scheduling to exploit these opportunities. The implementation instruments batch workloads at compile time with code that performs both monitoring and self-scheduling. The simple borrow-idle policy requires no change to the latency-critical workload. More aggressive policies require the latency-critical framework to expose the request queue length and a current request identifier via shared memory. Batch threads use `nanonap` to release hardware resources rapidly without relinquishing their SMT hardware context.

Our current design assumes an environment consisting of a single latency-critical workload, and any number of instrumented batch workloads. (Scheduling two or more distinct latency-critical services simultaneously on one server is a different and interesting problem that is beyond our scope.) Our instrumentation binds threads to cores with `setaffinity()` to force all request threads onto the identifiable request lane and batch threads onto partner batch lane(s). The underlying OS is free to sched-

ule batch threads on batch lanes. Each batch thread will then fall into a monitoring and self-scheduling rhythm.

5.3.1 Nanonap

This section introduces the system call `nanonap` to monitor and schedule threads at a fine granularity. The key semantics `nanonap` delivers is to put the hardware context to sleep *without* releasing the hardware to the OS scheduler. We first explain why existing mechanisms, such as `mwait`, `WRLOS`, and `hotplug` do not directly deliver the necessary semantics.

The `mwait` instruction releases the resources of a hardware context with low latency. This instruction is available in user-space on SPARC and is privileged on x86. The IBM PowerEN user-level `WRLOS` instruction has similar semantics [Meneghin et al., 2012]. Calls to `mwait` are normally paired with a `monitor` instruction that specifies a memory location that `mwait` monitors. The OS or another thread wakes up the sleeping thread by writing to the monitored location or sending it an interrupt. The Linux scheduler uses `mwait` to save energy. It assigns each core a privileged idle task when there are no ready tasks. Idle tasks call `mwait` to release resources, putting the hardware in to a low-power state. Unfortunately, simply building upon any of these mechanisms in *user space* is insufficient because the OS may, and is likely to, schedule other ready threads to the released hardware context. In contrast, because it disables preemption, `nanonap` ensures that no other thread runs on the lane, releasing all hardware resources to its partner lane.

Another mechanism that seems appealing, but does not work, is `hotplug`, which prohibits any task from executing in specified SMT lanes. The OS first disables interrupts, moves all threads in the lane(s), including the thread that invoked `hotplug`, to other cores, and switches the lane(s) to the idle task which then calls the `mwait` instruction. While `hotplug` moves threads off a lane to other cores, user-space calls such as `futex` yield the lane, so other threads may execute in it. Therefore, neither the `hotplug` interface nor user-space locking nor calls to `mwait` are designed to release and acquire SMT lanes to and from each other because a thread does not *retain exclusive ownership of a lane while it pauses*.

We design a new system call, `nanonap`, to control the SMT microarchitecture hardware resources directly. Any application that wants to release a lane invokes `nanonap`, which enters the kernel, disables preemption, and sleeps on a per-CPU `nanonap` flag. From the kernel's perspective, `nanonap` is a normal system call and it accounts for the thread as if the thread is still executing. Because `nanonap` does not disable interrupts and the kernel does not preempt the thread that invoked the `nanonap`, the SMT lane stays in a low-power sleep state until the OS wakes the thread up with an interrupt or the ELFEN scheduler sets the `nanonap` flag. After the SMT lane wakes up, it enables preemption and returns from the system call. Figure 5.6 shows the pseudocode of `nanonap`, which we implement as a wrapper that invokes a virtual device using the Linux OS's `ioctl` interface for devices.

No starvation or new security state The `nanonap` system call and monitoring of request lanes do not cause starvation or pose additional opportunities for security breaches. Starvation does not occur because `nanonap` does not disable interrupts. The scheduler may wake up any napping threads and schedule a different batch thread on the lane at the end of a time quanta, as usual. When a batch thread wakes up or a new one starts executing, it tests whether its request lane partner is occupied and if so, puts itself to sleep. Since the OS accrues CPU time to batch threads waiting due to a `nanonap`, user applications cannot perform a denial of service attack simply by continuously calling `nanonap`, since the OS will schedule a napping thread away after they exhaust their time slice.

The Elfen instrumentation monitors system state to make decisions. It reads memory locations and performance counters that reveal if the core has multiple threads executing. All of this system state is *already* available to threads on the same core — Elfen reveals no additional information about co-runners to each other.

5.3.2 Latency of Thread Control

This section presents an experiment that measures the latency of sleeping and waking up with `nanonap`, `mwait`, and `futex`. Measuring these latencies is challenging because detecting exactly when a lane releases hardware resources must be inferred, rather than measured directly.

When a batch thread executes `mwait` on our Intel hardware, the lane first enters the shallow sleeping C1 state immediately. If no other thread executes in the lane for a while, it then enters a deep sleep state and releases its hardware resources to the active request lane. We measure how long it takes the lane to enter the deep sleeping state indirectly as follows. The CPU executes a few μops to transition an SMT lane from the shallow to the deep sleep state. For measurement purposes, we thus configure the measurement thread to continuously record how many μops the measurement thread has retired and how many μops the whole core retires every 150 cycles. When the measurement thread notices that the sleeping SMT lane does not retire any μops for a while, then retires a few more μops, and then stops retiring μops, it infers that the SMT lane is in the deep sleep state.

Figure 5.7 shows a microbenchmark that measures the latencies of sleeping and waking up with `nanonap`, `mwait`, and `futex`. The microbenchmark has two threads: a measurement thread and another thread, T2. The measurement thread puts T2 to sleep and wakes it up. The two threads execute on the same core but different SMT lanes. The measurement thread sets a flag, forcing T2 to sleep (line 5). T2 then executes `sleep` which either calls `nanonap` or `futex` to put the SMT lane to sleep, according to which is being measured. The `wait_until_t2_goes_to_sleep()` (line 6) call performs the deep sleep detection process described in the above paragraph. We measure wake-up latency directly (lines 10 to 13). The measurement thread sets a flag (line 11) and then detects when T2 starts executing instructions (line 12).

We execute each configuration 100 times. Figure 5.8 presents the time and the 95% confidence interval for using `nanonap`, `mwait`, and `futex` to sleep and wake-up

```
/****************** USER ********************/
void nanonap() {
  ioctl(/dev/nanonap);
}
/****************** KERNEL ******************/
nanonap virtual device: /dev/nanonap;
per_cpu_variable: nap_flag;
ioctl(/dev/nanonap) {
  disable_preemption();
  my_nap_flag = this_cpu_flag(nap_flag);
  monitor(my_nap_flag);
  mwait();
  enable_preemption();
}
```

Figure 5.6: Pseudo code for `nanonap`.

```
/***** MEASUREMENT THREAD ON ONE SMT LANE *****/
void measure() {
  /* measure send-to-sleep latency */
  start_sleep_request = timestamp();
  ask_t2_sleep();
  wait_until_t2_goes_to_sleep();
  finish_sleep_request = timestamp();

  /* measure wake-up latency */
  start_wakeup_request = timestamp();
  wakeup_t2();
  wait_until_t2_wakes_up();
  finish_wakeup_request = timestamp();

  if (measuring_futex || measuring_nanonap) {
    sleep_latency =
      finish_sleep_request - start_sleep_request;
    wakeup_latency =
      finish_wakeup_request - start_wakeup_request;
  }
  if (measuring_mwait) {
    sleep_latency = finish_sleep_request - mwait_start;
    wakeup_latency = mwait_finish - start_wakeup_request;
  }
}
/************ T2 ON OTHER SMT LANE ************/
void sleep() {
  if (measuring_futex)
    wait_on_futex();
  else if (measuring_nanonap || measuring_mwait)
    nanonap();
}
void nanonap() {
  ...
  mwait_start = timestamp();
  monitor(flag);
  mwait();
  mwait_finish = timestamp();
  ...
}
```

Figure 5.7: Microbenchmark that measures time to sleep with `nanonap`, `mwait`, and `futex`.

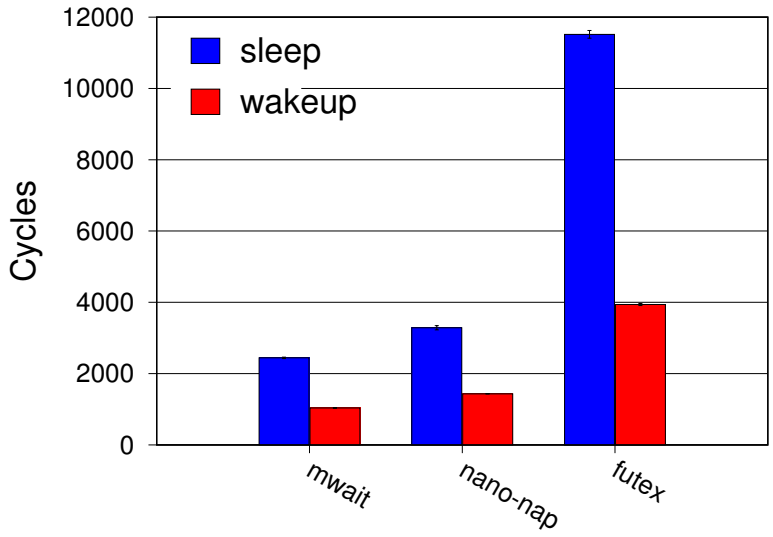


Figure 5.8: Time to sleep and wake-up SMT partner lanes.

a thread executing in a partner SMT lane. The time to put a lane to sleep for `mwait` is 2 443 cycles, is 3 285 cycles for `nanonap`, and is 11 518 cycles for `futex`, 3.5 times slower than `nanonap`. Waking up a lane directly with `mwait` takes 1 036 cycles — essentially the hardware latency of wake-up. The latency of `nanonap`'s wake-up is similar to `mwait`'s at 1438 cycles. However, `futex` takes 3 968 cycles, which is 2.72 times slower than `nanonap`. Although `futex` is substantially slower, this latency is likely tolerable, since most idle periods are more than 1 million cycles on our 2 GHz machine (1 ms in Figure 5.5). However, as explained above, neither the semantics of locks nor user-space calls to `mwait` are adequate for our purposes.

5.3.3 Continuous Monitoring and Signaling

Sleeping and waking up fast is necessary but not sufficient. The scheduler has to know when to act. We further exploit the `nanonap` mechanism to improve over our Shim [Yang et al., 2015a] fine-grain profiling tool. Shim views time-varying software and hardware events as continuous 'signals' (in the signal processing sense of the word). Rather than using interrupts to examine request threads, as many profiling tools do, we configure our batch threads to continuously read signals from memory locations and hardware performance counters to *profile* request threads. In the simplest case, the profiling observes whether the request thread is executing. Our prior work shows that Shim accurately observes events at granularities as fine as 100s to 1000s of cycles with overheads of a few percent when executing on another core. However, when threads share an SMT core, we saw similar overheads from SMT sharing as shown in Figure 5.4. In this chapter, we use the `nanonap` mechanism to

essentially eliminate this overhead.

Whereas Shim observes signals from a *dedicated thread*, here we (1) use GCC `-pg` instrumentation [GCC, 2016] to insert checks at method prologues into *C batch workloads* and (2) piggyback on the default Java VM checks at every method entry/exit and loop backedge [Lin et al., 2015]. These mechanisms add minimal overhead as shown by Lin et al. [2015] and, most importantly, remove the need for a third profiling thread to observe request threads.

At each check, the fast-path consists of a few instructions to check monitored signals. For efficiency, this fast path is inlined to the body of compiled methods. If the observed signal matches the condition (e.g., the scheduler sets the memory location that indicates the request lane is idle), the batch thread jumps to an out-of-line function to handle the task of putting itself to sleep.

5.3.4 Elfen Scheduling

We design and implement four policies that borrow underutilized resources without compromising SLOs.

Borrowing Idle Cycles The simplest way to improve utilization is to run the batch workload *only* when the latency-critical workload is idle. Section 5.2 analyzed the maximum CPU utilization of Lucene while meeting a practical SLO at ~70% of one SMT lane, which corresponds with prior analysis of latency-critical workloads [Dean and Barroso, 2013; Delimitrou and Kozyrakis, 2014; Ren et al., 2013; Haque et al., 2015; Hauswald et al., 2015]. Therefore even when the latency-sensitive workload is executing at the maximum utilization at which it can meet SLOs, there is an opportunity to improve utilization by 30% if the batch workload can borrow this excess capacity. At lower loads, there is even more opportunity.

This policy enforces mutual exclusion. Batch threads execute only when the request lane is empty. When a request starts executing, the batch thread immediately sleeps, relinquishing its hardware resources to the latency-critical request. When the request lane becomes idle, the batch thread wakes up and executes in the batch lane.

Figure 5.9(a) shows the simple modifications to the kernel and batch workloads required to implement this policy. We add an array (software channels) called `cpu_task_map` that maps a lane identifier to the current running task. At every context switch, the OS updates the map, as shown in `task_switch_to()`. By observing this signal, the scheduler knows which threads are executing in the SMT lanes. At each check, the scheduler determines whether the `idle_task` is executing in the request lane. If the request lane is idle, the scheduler either continues executing the batch thread in its lane or starts a batch thread. If the request lane is occupied, the scheduler immediately forces the batch thread to sleep with `nanonap`.

Fixed Budget Borrowing idle cycles is simple and as we show, effective, but we can further exploit underutilized resources when requests may incur some overhead and still meet their SLO. In particular, short requests, which typically dominate, easily

```
/****************** KERNEL ******************/
/* maps lane IDs to the running task */
exposed SHIM signal: cpu_task_map

task_switch(task T) { cpu_task_map[thiscpu] = T; }
idle_task() { // wake up any waiting batch thread
  update_nap_flag_of_partner_lane();
  ......
  mwait();
}
/***************** BATCH TASKS *****************/
/* fast path check injected into method body */
check:
if (!request_lane_idle) slow_path();

slow_path() { nanonap(); }
```

(a) Borrow idle policy.

```
/********** LATENCY CRITICAL WORKLOAD *********/
exposed SHIM signal: queue_len

/***************** BATCH THREADS ***************/
per_cpu_variable: lane_status = NORMAL;
per_cpu_variable: start_stamp;
check:
if (request_lane_idle && queue_len == 0) {
  lane_status = NEW_PERIOD;
} else if (!request_lane_idle) {
  slow_path();
}
slow_path() {
  switch (lane_status) {
  case NORMAL:
    nanonap();
    break;
  case NEW_PERIOD:
    lane_status = CO_RUNNING;
    start_stamp = rdtsc();
    break;
  case CO_RUNNING:
    now = rdtsc();
    if (now - start_stamp >= budget)  // expired
      lane_status = NORMAL;
} }
```

(b) Fixed budget policy.

Figure 5.9: The pseudocode of four scheduling policies (the borrow idle policy and the fixed budget policy).

```
/********** LATENCY CRITICAL WORKLOAD *********/
exposed SHIM signals: queue_len, running_request

/*************** BATCH THREADS ***************/
/* Same as the fixed budget policy, except... */
per_cpu_variable: last_request
...
 case NEW_PERIOD:
   ...
   last_request = running_request;
   ...
 case CO_RUNNING:
   if (running_request != last_request &&
       queue_len == 0) {
     last_request = running_request;
     start_stamp = rdtsc();
   }
...
```

(c) Refresh budget policy.

```
/*************** BATCH THREADS ***************/
/* Same as the refresh budget policy, except... */
...
 case CO_RUNNING:
   ...
   /* calculate IPC of LC lane */
   ratio = ref_IPC / (ref_IPC - LC_IPC)
   real_budget = budget * ratio;
   if (now - start_stamp >= real_budget)
     lane_status = NORMAL;
...
```

(d) Dynamic refresh policy.

Figure 5.9: The pseudocode of four scheduling policies (the refresh budget policy and the dynamic refresh policy).

meet the SLO under moderate loads. We consider the maximum slowdown requests can incur under a certain load as a budget for the batch workload. As an example, consider an SLO latency of 100 ms for 99% of requests. If 99% of requests executing exclusively on the core complete in 53 ms at some RPS, then there exists headroom of $100 - 53 = 47$ ms. We thus could take a budget of 47 ms for executing batch tasks. (We leave more sophisticated policies that also incorporate load along the lines of Haque et al. [2015] to future work.)

Given a budget, the fixed-budget scheduler will execute batch threads concurrently with requests in their respective SMT lanes when the scheduler is confident that the batch threads will not slow down any request longer than the given budget. Co-running with a request for T ms slows down the processing time of the request less than T ms. For requests that never wait in the queue, the processing time is the same as the request latency. So, it is safe to co-run with these requests for a budget period. Figure 5.9(b) shows the implementation of this policy. Line 7 detects when the request queue is empty and renews the budget period, such that the next request will co-execute with the batch thread for the fixed budget.

As we showed in Section 5.2, the request lane is frequently idle for short periods because after one request finishes there are no pending requests, and most requests are short. The fixed-budget scheduler only uses its budget when a new request that never waits in the queue starts executing in the request lane. When the scheduler detects that a new request starts executing and the `lane_status` is set to `NEW_PERIOD` because the request queue was empty before this requests started, it co-schedules the batch thread in its lane for the budget period. If the request is finished in the period and there are no waiting requests, the scheduler resets the budget and uses it for the next request. When the budget expires, the scheduler puts the batch thread to sleep. When another idle period begins because the request terminates, the request queue is empty, and no other request is executing, the scheduler restarts the batch thread and repeats this process. Note that if N requests execute in quick succession without idle gaps, this simple scheduler only co-executes the batch thread with the first request that begins after an idle period. This conservative strategy ensures that each request is only impacted for the budget period of its execution.

Refresh Budget The refresh budget policy extends the fixed budget policy based on the observation that once a request has completed *and* the queue is empty, the budget may be refreshed. The rationale is that the original budget was calculated based on avoiding a slowdown that could prevent the just-completed task from meeting the SLO. Once that task completes, then the budget may be recalculated with respect to the *new* task meeting the SLO. However, because the slowdown imposed by the batch workload is imparted not just on the running request, but on all requests behind it in the queue, we only refresh the budget if the task changes *and* the queue is empty. Figure 5.9(c) shows the code.

Dynamic Budget The dynamic budget policy is the most aggressive policy and builds upon the refresh budget policy. It uses a *dynamic budget* that is continuously

adjusted according to the base budget and the IPC of the latency-critical request. This policy requires us first to profile the IPC with no interference and then to monitor the impact of co-running on request IPC. We implement the monitoring based on the sampling ideas in SHIM [Yang et al., 2015a]. We read the IPC hardware performance counter of the request lane from the batch lane, at high frequency with low overhead. When the latency-critical request's IPC is high, it will be proportionately less affected by the batch workload, so we adjust the dynamic budget accordingly.

5.4 Methodology

Hardware & OS We use a 2.0 GHz Intel Xeon-D 1540 Broadwell [Intel, 2013b] processor with eight two-way SMT cores, a 12 MB shared L3. Each core has a private 256 KB L2, a 32 KB L1D and a 32 KB L1I. The TurboBoost maximum frequency is 2.6 Ghz, TDP is 45 W. The machine has 16 GB of memory and two Gigabit Ethernet ports. We disable deep sleep and TurboBoost.

We use Ubuntu 15.04, Linux version 4.3.0, and the perf subsystem to access the hardware performance counters. We implement the `nanonap` mechanism as a virtual device as shown in Figure 5.6. We modify the idle task to wake up sleeping batch lanes as shown in Figure 5.9(a). We expose a memory buffer to user space to determine which tasks are running on which cores.

Latency-Critical Workload We use the industrial-strength open-source Lucene framework to model behavior similar to the commercial Bing web search engine [Haque et al., 2015] and other documented latency-critical services [Dean and Barroso, 2013; Delimitrou and Kozyrakis, 2014; Ren et al., 2013; Hauswald et al., 2015]. Load variation results from both the number of documents that match a request and from ranking calculations. We considered using `memcached`, a key-value store application, because it is an important latency-critical workload for Facebook [Hart et al., 2012; Nishtala et al., 2013] and a popular choice in the OS and architecture communities. However, each request offers the same uniformly very low demand ($<$10 K instructions) [Hart et al., 2012], which means requests saturate the network before they saturate the CPU resources on many servers. Recent work offers OS and hardware solutions to these network scalability problems [Belay et al., 2014; Lim et al., 2014], which we believe if combined with our work would be complementary. We leave such investigations to future work.

We execute Lucene (svn r1718233) in the Open JDK 64 bit server VM (build 25.45-b02, mixed mode). We use the Lucene performance regression framework to build indexes of the first 10 M files from Wikipedia's English XML export [Wikipedia, 2016] and use 1141 term requests from `wikimedium.10M.nostopwords.tasks` as the search load. The indexes are small enough to be cached in memory on our machine. We warm up the server before running any experiments.

We send Lucene requests from another machine that has the same specifications as the server. The two machines are connected via an isolated Gigabit switch. For

each experiment, we perform 20 invocations. For each invocation, the client loads 1141 requests, shuffles the requests, and sends requests 5 times. The client issues search requests at random intervals following an exponential distribution around the prescribed RPS mean rate. We report the median result of the 20 invocations. The 95% confidence interval is consistently lower than ± 0.02.

Batch Workloads We use 10 real-world Java benchmarks from the DaCapo 2006 release [Blackburn et al., 2006a] and three micro C benchmarks, `Loop`, `Matrix`, and `Flush`. The DaCapo benchmarks are popular open-source applications with non-trivial memory loads that have active developers and users. Using DaCapo as batch workloads represents a real world setting. The C micro benchmarks demonstrate the generality of our approach and give us control over the interference pattern. `Loop` calls an empty function and has an IPC of 1. It consumes front-end pipeline resources. `Matrix` calls a function that multiplies a 5×5 matrix, a computationally intensive high IPC workload. It consumes both front-end pipeline and functional-unit resources. `Flush` calls a function that zeros a 32 KB buffer, a disruptive co-runner that flushes the L1D cache.

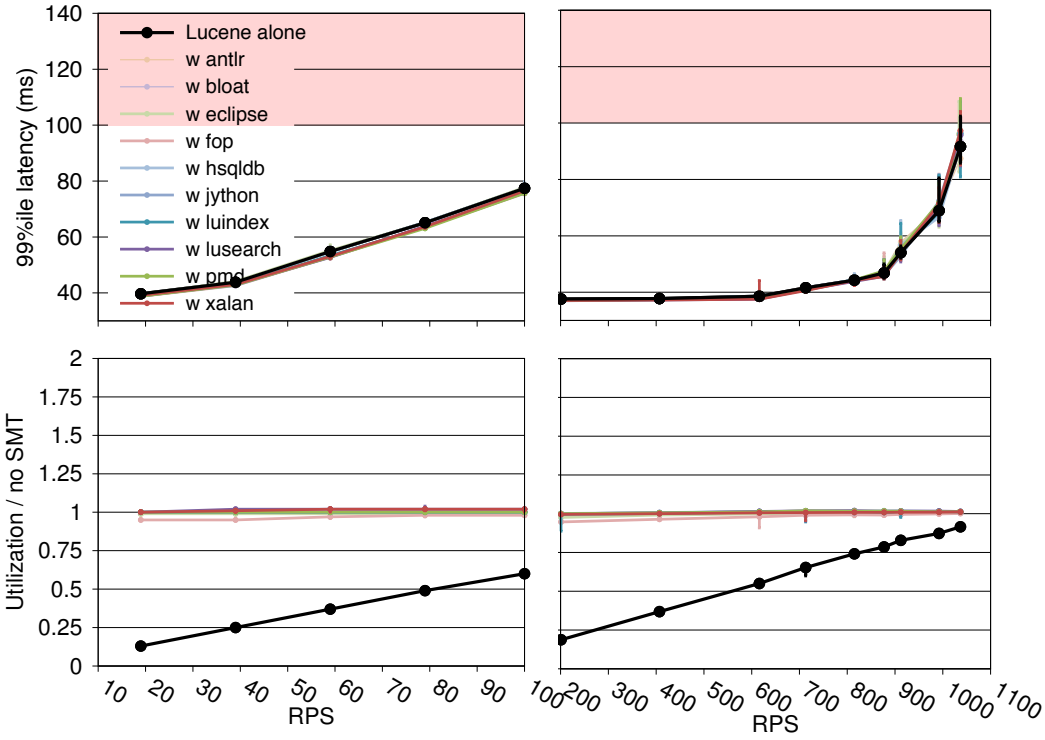
We run Java benchmarks with JikesRVM [Alpern et al., 2005], release 3.1.4 + git commit fd68163, a Java-in-Java high performance Virtual Machine, using a large 200 MB heap. The JIT compiler in JikesRVM already inserts checkpoints for thread control and garbage collection into function prologues, epilogues and loop back-edges. We add to these a check for co-runner state, as shown in Figure 5.9. For C micro benchmarks, we use GCC's `-pg` instrumentation option [GCC, 2016] to add checks to method prologues.

Measurements We use a target, 100 ms request latency for 99% of requests, as our SLO in all of our experiments, which is a practical SLO target for the search engine.

5.5 Evaluation

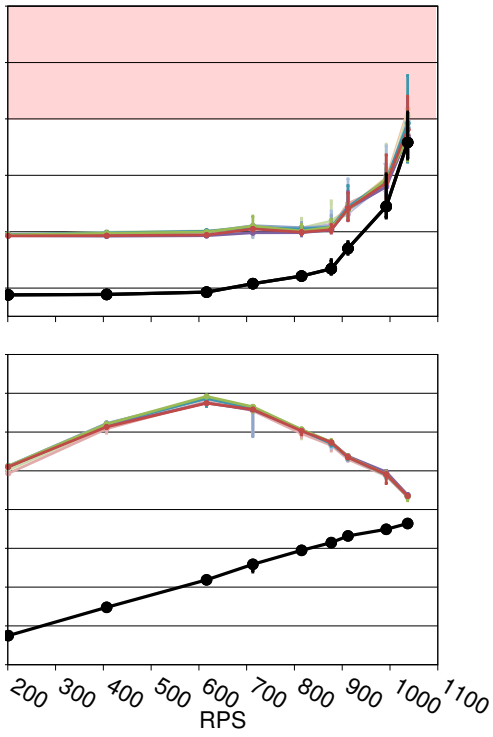
This section evaluates the ability of Elfen to improve server utilization while meeting Service Level Objectives (SLOs) and Elfen overheads.

Borrow idle We first present Elfen configurations that use the *borrow idle* policy with DaCapo as the batch workload. This policy minimizes the impact on request latencies. Figure 5.10(a) plots latency (top) and utilization (bottom) versus requests per second (RPS) on the x-axis for Lucene without (black) and with each of the ten DaCapo batch workloads (colors) executing on one two-way SMT core of the eight-core Broadwell CPU. Figure 5.10(b) presents these same configurations executing seven instances of each DaCapo benchmark on seven cores. The eighth core manages network communication (receiving requests and returning results), queuing, and starting requests for the latency-sensitive workload. We plot median latency; error bars indicate 10th and 90th percentiles.



(a) Borrow idle, one core

(b) Borrow idle, seven cores



(c) Dynamic refresh budget, seven cores

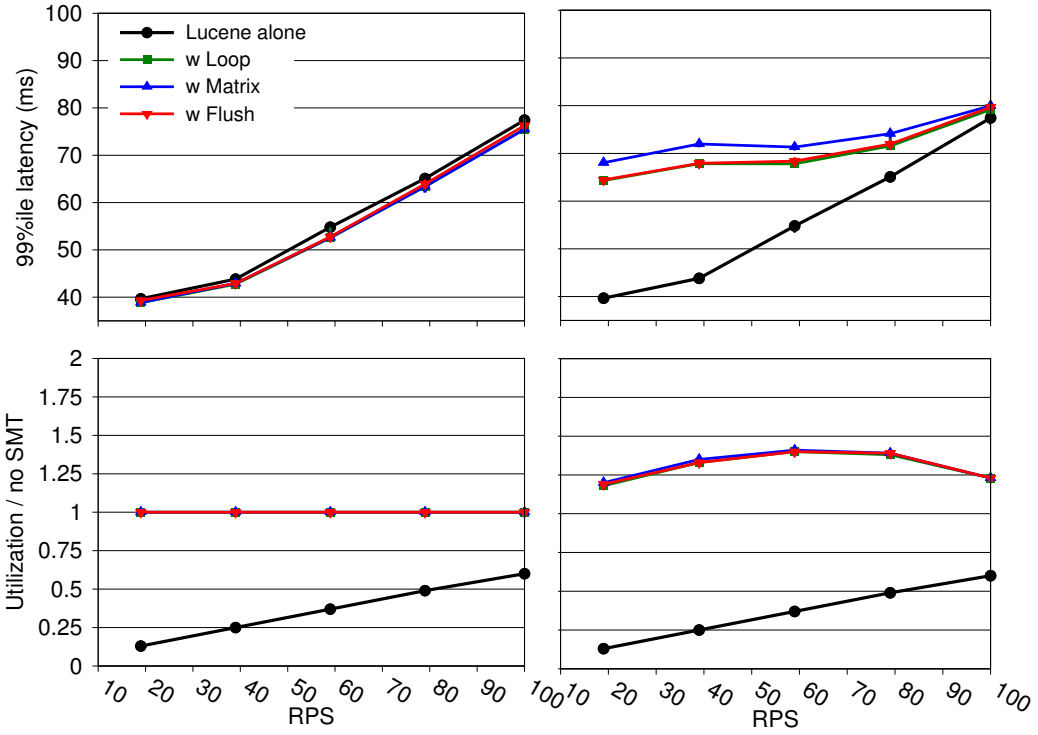
Figure 5.10: 99th percentile latency (top) and utilization (bottom) for Lucene co-running with DaCapo batch workloads.

The results in Figure 5.10(a) and 5.10(b) show that *executing these batch workloads in idle cycles imposes very little impact on Lucene's SLO on a single core or a CMP.* Elfen achieves essentially the same 99th percentile latency at the same requests per second (RPS) with or without batch execution. In fact on one core, Elfen sometimes delivers slightly lower latencies for Lucene when executing each of the batch workloads in the other lane during idle periods. This results occur because running the batch thread in the other lane causes the core never to enter any of its sleep states. When a new request arrives, the core is guaranteed not to be sleeping, its request lane is empty, and thus the core will service requests slightly faster. With the borrow idle policy, the peak utilization of the core is 100% out of 200% since each core has 2 hardware contexts, but by design, only one is active at a time. Because Elfen keeps the core busy, executing requests as they arrive in one lane and batch threads with mutual exclusion in the other, it often achieves its peak potential of 100% utilization, but when the utilization of the batch workload is low, the total utilization may be less than 100%.

The chip multiprocessor (CMP) results in Figure 5.10(b) show better throughput scaling than just a factor of seven. For example, at 60 ms, the single core system can sustain about 70 RPS, while the seven-core system can sustain as much as 1000 RPS. Remember that most requests are short, and long requests contribute most to tail latencies. CMPs better tolerate long request latencies than a single core by executing multiple other short requests on other cores, so fewer short requests incur queuing delay when a long request monopolizes a core. At moderate loads, we again see some improvements to request latency when co-running with batch workloads because the cores never sleep, whereas cores are sometimes idle long enough without co-runners to sleep. However, continuously and fully utilizing all seven cores on the chip incurs more interference, and thus we see some notable degradations in the 99%ile latency at high load. There are two sources of increased latency. First, the effects of managing the queue and request assignment, which shows some non-scalable results. For example, even small amounts of contention for the request queue impacts tail latency independent of Elfen. Elfen slightly exacerbates this problem. Second, as prior work has noted and addressed [Herdrich et al., 2013; Lo et al., 2015; Mars et al., 2011], requests and batch threads can contend for shared chip-level resources on CMPs, such as memory and bandwidth. Adding such techniques to Elfen should further improve its effectiveness.

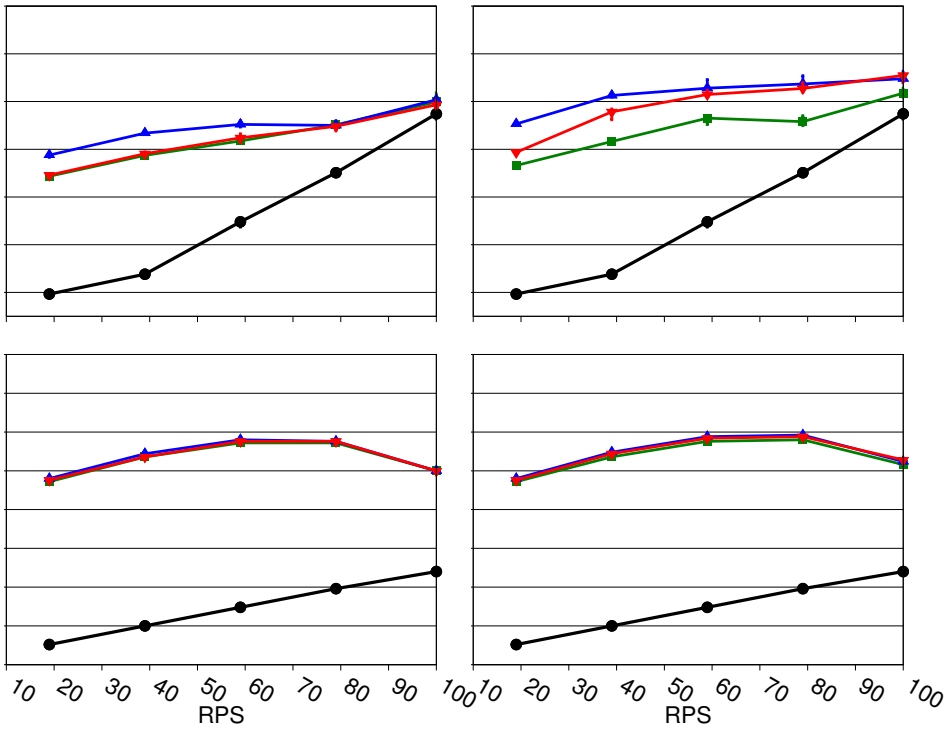
Increasing Utilization on a Budget Figure 5.11 presents latency (top graphs) and utilization (bottom) for the four Elfen scheduling policies described in Section 5.3: borrow idle, fixed budget, refresh budget, and dynamic refresh on one core. The budget-based policies all borrow idle cycles and trade latency for utilization, slowing the latency-critical requests to increase utilization. Comparing the top row in the figure shows that increasingly aggressive policies cause more degradations in the 99th percentile latency. In these RPS ranges, Lucene's requests meet the 100 ms SLO latency target, but are degraded.

Borrowing idle cycles and co-executing batch threads with requests increases



(a) Borrow idle

(b) Fixed budget



(c) Refresh budget

(d) Dynamic refresh

Figure 5.11: 99th percentile latency (top) and utilization (bottom) for Lucene co-running with C microbenchmarks under four ELFEN policies on a single two-way SMT core.

utilization significantly. Comparing across the utilization figures reveals that the budget-based policies further improve utilization compared to borrowing idle cycles. Core utilization rises as load increases. At moderate loads, Elfen achieves utilizations over 1.4 for the *fixed budget* policy and 1.5 for the *dynamic refresh* policy. All budget-based policies achieve utilizations of at least 1.2. When the system becomes highly loaded with requests, Elfen adjusts by executing the co-runners less, and thus total utilization drops. While all of the Elfen policies are effective at trading off utilization for SLOs, the most aggressive *dynamic refresh* policy consistently runs at higher utilization. The *dynamic refresh* policy is performing precise, fine grain monitoring of request IPC to more accurately and effectively manage this tradeoff. Although we study IPC, Elfen may monitor and partition other resources, such as memory and cache. Although higher utilization is appealing, some service providers may not be willing to sacrifice throughput of latency-critical tasks, so for them the most practical policy may be to borrow idle cycles.

Figure 5.10(c) shows the latency and utilization results for the most aggressive dynamic refresh policy on our CMP with DaCapo as the batch workload. This policy degrades the 99th percentile latency by 20 ms before reaching a peak utilization of 1.75 at around 600 RPS. At larger RPS, Elfen schedules the batch less, system utilization drops and the latency approaches to the same level of the borrow idle policy.

Overhead on Batch Workload Overhead on the batch workload comes from instrumentation, interference with the latency-sensitivity requests, and being frequently paused and restarted. As we pointed out above, Lin et al. [2015] show the instrumentation overheads are low, at most a couple percent.

Figure 5.12 measures these other overheads. It presents the execution time, user time utilization, and user level IPC of each DaCapo benchmark co-running with Lucene normalized to its execution alone on one core. When co-running, we use the borrow idle policy and load the Lucene at 80 RPS, which leads to about 50% utilization for both Lucene and each DaCapo benchmark. The execution time of co-running each DaCapo benchmark increases by 49% on average as predicted by the 50% utilization. There are small variations in these slowdowns, but none of them are due to DaCapo programs executing more instructions when co-running — the number of retired instruction at user level is the same. Furthermore, DaCapo does not execute instructions less efficiency, because IPC decreases are only 1%.

Variation in execution times is due to variations in utilization already present in the DaCapo benchmarks. If the batch workload is idle for some other reason (e.g., waiting on I/O or a lock), then a request that forces it to stop executing will affect it less. The more idle periods the batch workload has, the less execution is degraded. This effect causes normalized execution time and utilization to be strongly correlated. For instance, the *pmd* benchmark incurs the largest slowdown in execution time, 59%, and the largest utilization reduction, 36%. The *fop* benchmark has the lowest native utilization in these benchmarks. Consequently, it has both the smallest slowdown and the smallest utilization reduction, 47% and 26%.

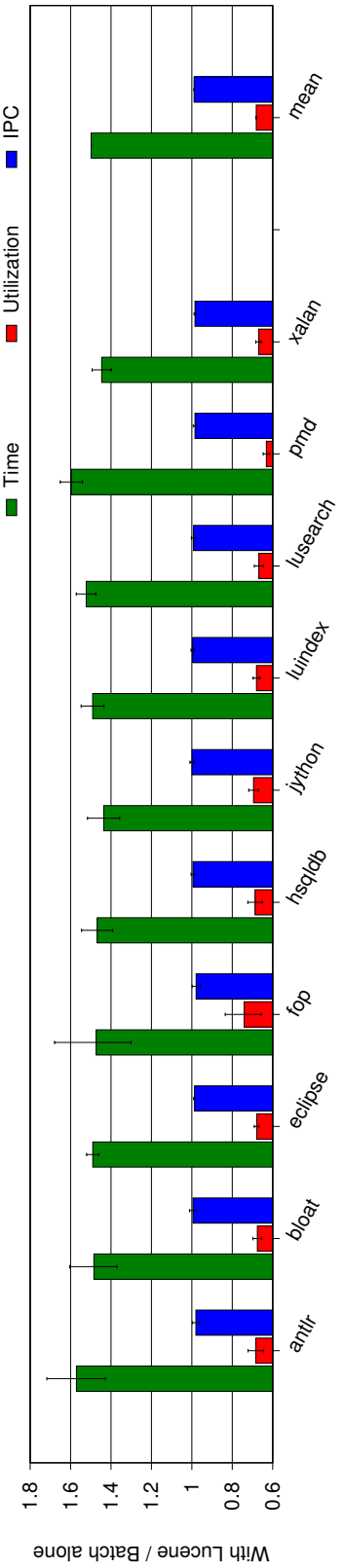


Figure 5.12: Normalized DaCapo benchmark execution time, user space CPU utilization, and IPC.

5.6 Related Work

Exploiting SMT The queuing delay caused by load spikes results in highly non-linear increases in tail-latency. To handle the load spikes, prior work [Haque et al., 2017, 2015; Hsu et al., 2015; Ren et al., 2013; Kasture et al., 2015] explores SMT, DVFS, heterogeneous hardware, or parallelism for servers that exclusively handle latency-sensitive requests (no co-location). Elfen is largely orthogonal to these approaches since our work focuses on improving server utilization and mitigating the impact of co-running batch jobs.

Lo et al. [2015] demonstrate that naively co-running batch workloads with latency-critical workloads violates Google's SLO, even under light load. They show that for many latency-critical workloads, uncontrolled interference due to SMT co-location is unacceptably high, and conclude that it is not practical to share cores with SMT. Our results contradict this conclusion.

Herdrich et al. [2013] note that achieving latency objectives with current SMT hardware is challenging because the shared resources introduce contention that make it hard to reason about and meet SLOs. They propose SMT rate control as a building block to improve fairness and determinism in SMT, which dynamically partitions resources and implements biased scheduling. These mechanisms should help limit interference on requests and complement our approach. They do not evaluate latency-critical workloads, seek to borrow idle cycles, or offer a fine-grain thread-switching mechanism, as we do here.

Inspired by our appraoch, Margaritov et al. [2019] designs a similar fine-grain thread-switching mechanism in hardware. They utilizes the mechanism to balance QoS and throughput of co-located latency-critical and batch workloads on SMT cores.

Accommodating Overheads Zhang et al. [2014] use offline profiling of batch workloads to precisely predict the overhead due to co-running with latency-critical requests on SMT. They then carefully provision resources to co-run batch workloads whilst maintaining SLOs for latency-critical workloads. Unlike our work, they do not attempt to minimize the overhead of co-running batch workloads. Rather, they predict and then accommodate it. They measured interference due to co-run batch workloads in the range of 30%-50%.

POSIX Real-Time Scheduling Leverich and Kozyrakis [2014] propose using POSIX real-time scheduling disciplines to prioritize requests over co-run batch threads. When hardware contexts are scarce, this approach ensures that latency-critical requests have priority — batch threads will be the first to block. When given sufficient hardware contexts however, the approach does not control for interference due to co-running. Thus it does not address the problem we address here: avoiding interference due to co-running while utilizing SMT.

Exposing and Evaluating `mwait` Anastopoulos and Koziris [2008] use `mwait` to release resources to another SMT thread when waiting on a lock. Wamhoff et al. [2014]

make `mwait` user-level visible and then use it to put cores into sleep states so as to provide power headroom for DVFS to boost performance on other cores which are executing threads on the program's critical path. They measure the latency of putting an entire core into a C1 sleep state on an Intel Haswell 4 770 and found that it was 4 655 cycles. This result is broadly consistent with our measurements, which are for a single hardware context on a more recent processor. With regard to semantics, Meneghin et al. [2012] claim fine-grain thread communication requires user-level mechanisms, whereas we offer an intermediate point that involves the OS, but not the OS scheduler. None of this prior work has the same semantics as `nanonap` for hardware control, which we exploit for both fine-grain monitoring and scheduling.

5.7 Summary

This chapter shows how to implement a Shim-inspired optimization. We show how Elfen uses SMT to execute latency-critical and batch workloads on the same server to increase utilization *without* degrading the SLOs of the latency-critical workloads. We show, given a budget, how to control latency degradations to further increase utilization while meeting SLOs. Our policies borrow idle cycles and control interference by reserving one lane for requests and one for batch threads. By reserving SMT lanes, Elfen always immediately executes the next request when the previous one completes or a new one arrives. Using Shim's high-frequency monitoring and `nanonap`, Elfen responds promptly to release core resources to requests or to control interference from batch threads. Our principled borrowing approach is extremely effective at increasing core utilization. Whereas current systems achieve utilizations of 5% to 35% of a 2-way core (by only using one lane at 10% to 70%) while meeting SLOs, Elfen's *borrow idle* policy uses both lanes to improve utilization at low load by 90% and at high load by 25%, delivering consistent and full utilization of a core at the same SLO. On CMPs, Elfen with the borrow idle policy is extremely effective as well, achieving its peak utilization without degrading SLOs for all but the highest loads. No prior work has managed this level of consistent server utilization without degrading SLOs.

Chapter 6

Conclusion

In their Turing Lecture, entitled "A New Golden Age for Computer Architecture", Hennessy and Patterson [2019] highlight that with the end of Moore's Law and Dennard Scaling, the future of performance improvement is in inventing full-stack optimizations. However, given the complexity of computer systems, discovering full-stack optimizations is a challenging discovery process: engineers must observe system behavior, develop insights, and design optimizations. This thesis empower engineers with a new observation tool that gives them high-frequency high-fidelity profiling information. It shows how to use this information to observe, analyze, and optimize the fine-grain behavior of production systems on commodity multicore hardware.

To observe the fine-grain behavior of production systems, we design Shim, a new continuous profiling approach working on today's multicore processors. Shim views computer systems as high-frequency signal generators. It samples software and hardware signals at high frequencies from observer threads on multicore hardware. We show that Shim filters out noisy samples efficiently using double-time error correction. We present case studies demonstrating that Shim delivers high-frequency high-fidelity signals. Shim improves sampling frequencies by three to five orders of magnitude, fundamentally altering which system behaviors are observable.

To analyze and control the fine-grain behavior of latency-critical systems, we design Tailor, a real-time latency controller that consists of two parts: 1) a Shim-based high-frequency profiler which continuously analyzes the fine-grain interactions of complex system components and promptly acts on hazardous system behaviors, 2) and an application-level network proxy which uses local node server redundancy to reduce the impact of unavoidable random hazardous behaviors. We present a case study demonstrating that Tailor not only diagnoses root causes of slow latency-critical requests, but also reduces tail latency by nine times via adjusting system configurations and mitigating the impact of unavoidable random JVM pauses with loca-node server redundancy.

To demonstrate that fine-grain control of system components leads to a new class of online profile-guided optimizations, we design Elfen, a Shim-based job scheduler that controls fine-grain interactions between latency-critical requests and batch jobs. We show that Elfen improves server utilization significantly without degrading the SLOs of the latency-critical requests by co-running batch jobs and latency-critical

requests on the same core but on different SMT contexts. We show that co-running batch jobs can use the Shim profiling approach to continuously monitor the status of latency-critical requests, and promptly release CPU resources to the paired latency-critical SMT context with the `nanonap` system call, which we introduce. Elfen opens a new way to substantially improve the utilization of datacenter running latency-critical workloads.

Today, gigahertz multicore processors are prevalent. This thesis enables online high-frequency profiling on these platforms and demonstrates that not only does high-frequency profiling discover full-stack optimization opportunities, but also can be used to implement new optimizations.

6.1 Future Work

This section highlights three future research directions that follow from this thesis: profiling distributed systems, designing customized profiling cores, and exploring more system signal processing techniques.

6.1.1 Distributed High-Frequency Profiling

In the era of cloud computing, developers need tools to help them observe, analyze, and control the fine-grain behavior of complex distributed systems. Shim could be extended to address this problem: On each node, we deploy a continuous high-frequency profiler, which in addition to sampling local hardware and software signals, continuously communicates with other remote profilers to draw a map showing fine-grain interactions between distributed nodes.

6.1.2 A Profiling Core

In this thesis, we repurpose existing multicore hardware to execute high-frequency profiling observers. It is wasteful to delicate a general-purpose core to high-frequency profiling, both in terms of power and CPU utilization. The general-purpose core also has limited channels with which to observe hardware events from other hardware contexts, and limited capabilities to control other hardware behavior. A customized core with the changes proposed in Section 3.1 could reduce the overhead of continuous high-frequency profiling, enlarge the scope of observable hardware and software events, and reduce the risk factor of leaking sensitive signals to general-purpose cores.

6.1.3 System Signal Processing

Processing high-frequency signals of system behavior is challenging. In this thesis, we introduce a few signal processing techniques: Shim uses the double-time error correction to improve the fidelity of signals, Tailor tracks latency-critical requests with TaleChain signals, and Elfen listens to signals of latency-critical requests and takes real-time scheduling actions. However, many more signal processing techniques

can be explored within the framework provided by this thesis such as indexing and searching large-volume real-time signal data, as well as new real-time signal filters and analyzers.

6.2 Final Words

Claude Shannon, the father of information theory, recalled a conversation with Alan Turing when he visited Turing's Laboratory in 1950:

> So I asked him what he was doing. And he said he was trying to find a way to get better feedback from a computer so he would know what was going on inside the computer. And he'd invented this wonderful command. See, in those days they were working with individual commands. And the idea was to discover good commands. And I said, what is the command? And he said, the command is put a pulse to the hooter, put a pulse to the hooter. Now let me translate that. A hooter is an English, in England is a loudspeaker. And by putting a pulse to it, it would just be put a pulse to a hooter. Now what good is this crazy command? Well, the good of this command is that if you're in a loop you can have this command in that loop and every time it goes around the loop it will put a pulse in and you will hear a frequency equal to how long it takes to go around that loop. And then you can put another one in some bigger loop and so on. And so you'll hear all of this coming on and you'll hear this "boo boo boo boo boo boo," and his concept was that you would soon learn to listen to that and know whether when it got hung up in a loop or something else or what it was doing all this time, which he'd never been able to tell before.
>
> From the oral history by Price [1982]

Shannon and Turing's interaction shows that from the very beginning of computing, people needed ways to understand what computations and computers were doing. They saw that computation could be interpreted as generating signals and understood as signal processing. In essence, this thesis views system behavior from a digital signal processing perspective. Our approach listens to the "sound" of computation by sampling software and hardware signals at orders of magnitude higher frequencies while limiting observer effects. It shows examples of how this increase in sampling rate leads to new analysis, insights, and optimizations. We believe it presages even more such opportunities.

Bibliography

Airbnb, 2018. Hypernova: A Service for Server-Side Rendering Your JavaScript Views. https://github.com/airbnb/hypernova. (cited on page 51)

Alpern, B.; Augart, S.; Blackburn, S. M.; Butrico, M.; Cocchi, A.; Cheng, P.; Dolby, J.; Fink, S. J.; Grove, D.; Hind, M.; McKinley, K. S.; Mergen, M.; Moss, J. E. B.; Ngo, T.; Sarkar, V.; and Trapp, M., 2005. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44, 2 (2005), 399–418. (cited on pages 33 and 94)

AMD, 2019. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 00h-0Fh Processors, accessed 05/2019. https://developer.amd.com/resources/developer-guides-manuals/. (cited on pages 10 and 16)

Ammons, G.; Ball, T.; and Larus, J. R., 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 85–96. http://doi.acm.org/10.1145/258915.258924. (cited on pages 16, 20, 24, and 49)

Anastopoulos, N. and Koziris, N., 2008. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 1–8. (cited on page 100)

Anderson, J. M.; Berc, L. M.; Dean, J.; Ghemawat, S.; Henzinger, M. R.; Leung, S.-T. A.; Sites, R. L.; Vandevoorde, M. T.; Waldspurger, C. A.; and Weihl, W. E., 1997. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 15, 4 (Nov. 1997), 357–390. (cited on pages 16, 21, 31, 47, and 49)

Apache Lucene, 2014. http://lucene.apache.org/. (cited on pages 51, 77, and 78)

ARM, 2019. CoreSight Architecture Overview, accessed on 05/2019. https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture. (cited on page 16)

Arnold, M.; Fink, S. J.; Grove, D.; Hind, M.; and Sweeney, P., 2000. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 47–65. Minneapolis, MN. (cited on page 33)

Arnold, M. and Grove, D., 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the International Symposium on Code*

Generation and Optimization, CGO '05, 51–62. IEEE Computer Society, Washington, DC, USA. doi:10.1109/CGO.2005.9. http://dx.doi.org/10.1109/CGO.2005.9. (cited on page 15)

Bartlett, J. F., 1981. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81 (Pacific Grove, California, USA, 1981), 22–29. ACM, New York, NY, USA. doi:10.1145/800216.806587. http://doi.acm.org/10.1145/800216.806587. (cited on page 72)

Belay, A.; Prekas, G.; Klimovic, A.; Grossman, S.; Kozyrakis, C.; and Bugnion, E., 2014. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 49–65. (cited on page 93)

Bertran, R.; Sugawara, Y.; Jacobson, H. M.; Buyuktosunoglu, A.; and Bose, P., 2013. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. *IBM Journal of Research and Development*, 57, 1/2 (Jan 2013), 4:1–4:17. http://dx.doi.org/10.1147/JRD.2012.2227580. (cited on page 22)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006a. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA, Oct. 2006), 169–190. (cited on pages 33 and 94)

Blackburn, S. M.; Hirzel, M.; Garner, R.; and Stefanović, D., 2006b. pjbb2005: The pseudojbb benchmark. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005. (cited on page 33)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 22–32. Tuscon, AZ. (cited on page 33)

Bond, M. D. and McKinley, K. S., 2007. Probabilistic calling context. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 97–112. Montreal, Quebec, Canada. (cited on page 16)

Boneti, C.; Cazorla, F. J.; Gioiosa, R.; Buyuktosunoglu, A.; Cher, C.-Y.; and Valero, M., 2008. Software-controlled priority characterization of POWER5 processor. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 415–426. (cited on pages 22 and 36)

Cantrill, B. M.; Shapiro, M. W.; and Leventhal, A. H., 2004. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX*

Annual Technical Conference, ATEC '04 (Boston, MA, 2004), 2–2. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1247415.1247417. (cited on page 16)

Chen, S.; Falsafi, B.; Gibbons, P. B.; Kozuch, M.; Mowry, T. C.; Teodorescu, R.; Ailamaki, A.; Fix, L.; Ganger, G. R.; Lin, B.; and Schlosser, S. W., 2006. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06 (San Jose, California, 2006), 63–65. ACM, New York, NY, USA. doi:10.1145/1181309.1181319. http://doi.acm.org/10.1145/1181309.1181319. (cited on page 16)

Cmelik, B. and Keppel, D., 1994. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, Tennessee, USA, 1994), 128–137. http://doi.acm.org/10.1145/183018.183032. (cited on page 49)

Conte, T. M.; Menezes, K. N.; and Hirsch, M. A., 1996. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29 (Paris, France, 1996), 36–45. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=243846.243855. (cited on page 16)

Dean, J. and Barroso, L. A., 2013. The tail at scale. *Communications of the ACM*, 56, 2 (2013), 74–80. (cited on pages 61, 72, 75, 78, 89, and 93)

DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; and Vogels, W., 2007. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 205–220. (cited on page 75)

Dehnert, J. C.; Grant, B. K.; Banning, J. P.; Johnson, R.; Kistler, T.; Klaiber, A.; and Mattson, J., 2003. The Transmeta Code Morphing^{TM} software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 15–24. (cited on pages 15 and 50)

Delimitrou, C. and Kozyrakis, C., 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 127–144. (cited on pages 75, 76, 78, 79, 89, and 93)

Demme, J. and Sethumadhavan, S., 2011. Rapid identification of architectural bottlenecks via precise event counting. In *IEEE/ACM Annual International Symposium on Computer Architecture* (San Jose, California, USA, 2011), 353–364. http://doi.acm.org/10.1145/2000064.2000107. (cited on pages 20, 24, and 49)

Elastic, 2019. Elasticsearch: The heart of the elastic stack. https://www.elastic.co/products/elasticsearch. (cited on page 51)

Facebook, 2018. React: A javascript library for building user interfaces. https://github.com/facebook/react. (cited on page 51)

GCC, 2016. Program instrumentation. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html. (cited on pages 89 and 94)

Graham, S. L., 1980. Table-driven code generation. *Computer*, 13, 8 (Aug 1980), 25–34. doi:10.1109/MC.1980.1653744. (cited on page 15)

Graham, S. L.; Kessler, P. B.; and Mckusick, M. K., 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82 (Boston, Massachusetts, USA, 1982), 120–126. ACM, New York, NY, USA. doi:10.1145/800230.806987. http://doi.acm.org/10.1145/800230.806987. (cited on page 15)

Gray, J., 1985. Why do computers stop and what can be done about it? (cited on page 72)

Ha, J.; Arnold, M.; Blackburn, S. M.; and McKinley, K. S., 2009. A concurrent dynamic analysis framework for multicore hardware. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 155–174. (cited on pages 20 and 49)

Haque, M. E.; He, Y.; Elnikety, S.; Nguyen, T. D.; Bianchini, R.; and McKinley, K. S., 2017. Exploiting heterogeneity for tail latency and energy efficiency. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 625–638. doi:10.1145/3123939.3123956. https://doi.org/10.1145/3123939.3123956. (cited on pages 72 and 100)

Haque, M. E.; hun Eom, Y.; He, Y.; Elnikety, S.; Bianchini, R.; and McKinley, K. S., 2015. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 161–175. (cited on pages 33, 72, 77, 78, 89, 92, 93, and 100)

Hart, S.; Frachtenberg, E.; and Berezecki, M., 2012. Predicting memcached throughput using simulation and modeling. In *Symposium on Theory of Modeling and Simulation (IMS/DEVS)*, 40:1–8. (cited on page 93)

Hauswald, J.; Laurenzano, M. A.; Zhang, Y.; Li, C.; Rovinski, A.; Khurana, A.; Dreslinski, R. G.; Mudge, T.; Petrucci, V.; Tang, L.; and Mars, J., 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 223–238. (cited on pages 78, 89, and 93)

Hauswirth, M.; Sweeney, P. F.; Diwan, A.; and Hind, M., 2004. Vertical profiling: Understanding the behavior of object-priented applications. *SIGPLAN Not.*, 39, 10 (Oct. 2004), 251–269. doi:10.1145/1035292.1028998. http://doi.acm.org/10.1145/1035292.1028998. (cited on page 16)

He, Y.; Elnikety, S.; Larus, J.; and Yan, C., 2012. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, Article 12: 1–14. (cited on page 75)

Hennessy, J. L. and Patterson, D. A., 2019. A new golden age for computer architecture. *Commun. ACM*, 62, 2 (Jan. 2019), 48–60. doi:10.1145/3282307. http://doi.acm.org/10.1145/3282307. (cited on page 103)

Herdrich, A.; Illikkal, R.; Iyer, R.; Singhal, R.; Merten, M.; and Dixon, M., 2013. SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms. In *HotPar*, 219–230. (cited on pages 96 and 100)

Hsu, C.; Zhang, Y.; Laurenzano, M. A.; Meisner, D.; Wenisch, T. F.; Mars, J.; Tang, L.; and Dreslinski, R. G., 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 271–282. doi:10.1109/HPCA.2015.7056039. https://doi.org/10.1109/HPCA.2015.7056039. (cited on pages 72, 83, and 100)

Intel, 2013a. Intel Core i7-4770 processor, 8m cache, 3.90 GHz. http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz. (cited on page 33)

Intel, 2013b. Intel Processor D-1540, 12M Cache, 2.00 GHz. http://ark.intel.com/products/87039/Intel-Xeon-Processor-D-1540-12M-Cache-2_00-GHz. (cited on page 93)

Intel, 2014. VTune Amplifier, accessed 11/2014. https://software.intel.com/en-us/intel-vtune-amplifier-xe/details. (cited on pages 3, 16, 19, 47, and 49)

Intel, 2015. Intel Processor D-1541, 12M Cache, 2.10 GHz. http://ark.intel.com/products/87039/Intel-Xeon-Processor-D-1540-12M-Cache-2_00-GHz. (cited on page 64)

Intel, 2019. Intel® 64 and ia-32 architectures software developer manuals, accessed 05/2019. https://software.intel.com/en-us/articles/intel-sdm. (cited on pages xiii, 10, 11, and 16)

iovisor, 2019. BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. (cited on page 16)

Jalaparti, V.; Bodik, P.; Kandula, S.; Menache, I.; Rybalkin, M.; and Yan, C., 2013. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, 219–230. (cited on page 83)

Kaffes, K.; Chong, T.; Humphries, J. T.; Belay, A.; Mazières, D.; and Kozyrakis, C., 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes. (cited on page 72)

Kaler, T.; He, Y.; and Elnikety, S., 2017. Optimal reissue policies for reducing tail latency. In *ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, 195–206. doi:10.1145/3087556.3087566. https://doi.org/10.1145/3087556.3087566. (cited on page 72)

Kanev, S.; Darago, J. P.; Hazelwood, K.; Ranganathan, P.; Moseley, T.; Wei, G.-Y.; and Brooks, D., 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15 (Portland, Oregon, 2015), 158–169. ACM, New York, NY, USA. doi:10.1145/2749469.2750392. http://doi.acm.org/10.1145/2749469.2750392. (cited on pages 16, 17, and 75)

Kannan, R. S.; Subramanian, L.; Raju, A.; Ahn, J.; Mars, J.; and Tang, L., 2019. GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks. In *ACM European Conference on Computer Systems (Eurosys)*, 34:1–34:16. doi:10.1145/3302424.3303958. https://doi.org/10.1145/3302424.3303958. (cited on page 72)

Kasture, H.; Bartolini, D. B.; Beckmann, N.; and Sanchez, D., 2015. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48 (Waikiki, Hawaii, 2015), 598–610. ACM, New York, NY, USA. doi:10.1145/2830772.2830797. http://doi.acm.org/10.1145/2830772.2830797. (cited on pages 72 and 100)

Kay, A., 2019. Alan Kay's answer to a Quora question titled "Was the Xerox Alto a prototype or a finished product?", accessed 05/2019. https://www.quora.com/Was-the-Xerox-Alto-a-prototype-or-a-finished-product-1. (cited on page 1)

Kim, S.; He, Y.; Hwang, S.-W.; Elnikety, S.; and Choi, S., 2015. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 7–16. (cited on page 83)

Knuth, D. E., 1971. An empirical study of fortran programs. In *Software: Practice and Experience*, 105–133. doi:10.1002/spe.4380010203. (cited on page 15)

Larus, J. R. and Ball, T., 1994. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24, 2 (Feb. 1994), 197–218. doi:10.1002/spe.4380240204. http://dx.doi.org/10.1002/spe.4380240204. (cited on page 16)

Leverich, J. and Kozyrakis, C., 2014. Reconciling high server utilization and sub-millisecond quality of service. In *ACM European Conference on Computer Systems (Eurosys)*, Article 4:1–14. (cited on pages 70 and 100)

Li, J.; Sharma, N. K.; Ports, D. R. K.; and Gribble, S. D., 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *ACM Symposium on Cloud Computing (SOCC)*, 9:1–9:14. doi:10.1145/2670979.2670988. https://doi.org/10.1145/2670979.2670988. (cited on page 70)

Li, S.; Lim, H.; Lee, V. W.; Ahn, J. H.; Kalia, A.; Kaminsky, M.; Andersen, D. G.; O, S.; Lee, S.; and Dubey, P., 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, 476–488. doi:10.1145/2749469.2750416. https://doi.org/10.1145/2749469.2750416. (cited on page 70)

Lim, H.; Han, D.; Andersen, D. G.; and Kaminsky, M., 2014. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 429–444. (cited on page 93)

Lin, Y.; Wang, K.; Blackburn, S. M.; Norrish, M.; and Hosking, A. L., 2015. Stop and Go: Understanding yieldpoint behavior. In *ACM International Symposium on Memory Management (ISMM)*, 70–80. (cited on pages 34, 89, and 98)

Linux, 2014a. Linux kernel profiling with perf, accessed 11/2014. https://perf.wiki.kernel.org/index.php/Tutorial#Event-based_sampling_overview. (cited on pages 3, 16, 19, 20, 24, 47, and 49)

Linux, 2014b. Perf core.c perf_duration_warn, accessed 11/2014. http://lxr.free-electrons.com/source/kernel/events/core.c#L229. (cited on pages 5, 20, and 49)

Lo, D.; Cheng, L.; Govindaraju, R.; Ranganathan, P.; and Kozyrakis, C., 2015. Heracles: improving resource efficiency at scale. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, 450–462. doi:10.1145/2749469.2749475. https://doi.org/10.1145/2749469.2749475. (cited on pages 72, 76, 79, 96, and 100)

Lorch, J. R. and Smith, A. J., 2001. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 50–61. (cited on page 83)

Luk, C.; Cohn, R. S.; Muth, R.; Patil, H.; Klauser, A.; Lowney, P. G.; Wallace, S.; Reddi, V. J.; and Hazelwood, K. M., 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 190–200. (cited on pages 16 and 49)

Margaritov, A.; Gupta, S.; González-Alberquilla, R.; and Grot, B., 2019. Stretch: Balancing qos and throughput for colocated server workloads on SMT cores. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, 15–27. doi:10.1109/HPCA.2019.00024. https://doi.org/10.1109/HPCA.2019.00024. (cited on page 100)

Mars, J.; Tang, L.; Hundt, R.; Skadron, K.; and Soffa, M. L., 2011. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations.

In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 248–259. (cited on pages 76 and 96)

McCandless, M., 2019. Lucene nightly benchmarks. https://people.apache.org/~mikemccand/lucenebench. (cited on page 53)

Meneghin, M.; Pasetto, D.; Franke, H.; Petrini, F.; and Xenidis, J., 2012. Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 131–132. http://researcher.watson.ibm.com/researcher/files/ie-pasetto_davide/PerfLocksQueues.pdf. Extended version at url, accessed May 2016. (cited on pages 85 and 101)

Mytkowicz, T.; Diwan, A.; Hauswirth, M.; and Sweeney, P. F., 2010. Evaluating the accuracy of Java profilers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 187–197. (cited on pages 16, 21, 24, 31, and 49)

Nethercote, N. and Seward, J., 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 89–100. San Diego, CA. (cited on page 49)

Nishtala, R.; Fugal, H.; Grimm, S.; Kwiatkowski, M.; Lee, H.; Li, H. C.; McElroy, R.; Paleczny, M.; Peek, D.; Saab, P.; Stafford, D.; Tung, T.; and Venkataramani, V., 2013. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 385–398. (cited on page 93)

Novaković, D.; Vasić, N.; Novaković, S.; Kostić, D.; and Bianchini, R., 2013. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX Annual Technical Conference (USENIX ATC)*, 219–230. (cited on page 76)

Nowak, A. and Bitzes, G., 2014. The overhead of profiling using PMU hardware counters. doi:10.5281/zenodo.10800. http://dx.doi.org/10.5281/zenodo.10800. (cited on page 49)

OProfile, 2014. OProfile, accessed 11/2014. http://oprofile.sourceforge.net. (cited on pages 16 and 47)

Pettersson, M., 2003. Linux Intel/x86 performance counters. http://user.it.uu.se/mikpe/linux/perfctr/. (cited on pages 24 and 49)

Price, R., 1982. Interview with claude shannon, oral-history web document, accessed 05/2019. https://ethw.org/Oral-History:Claude_E._Shannon. (cited on page 105)

Ren, G.; Tune, E.; Moseley, T.; Shi, Y.; Rus, S.; and Hundt, R., 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68. (cited on page 16)

Ren, S.; He, Y.; Elnikety, S.; and McKinley, K. S., 2013. Exploiting processor heterogeneity in interactive services. In *ACM International Conference on Autonomic Computing (ICAC)*, 45–58. (cited on pages 72, 78, 89, 93, and 100)

Rostedt, S., 2019. ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. (cited on page 16)

Satterthwaite, E., 1972. Debugging tools for high level languages. In *Software: Practice and Experience*, 197–217. doi:10.1002/spe.4380020303. (cited on page 15)

Schöne, R.; Molka, D.; and Werner, M., 2015. Wake-up latencies for processor idle states on current x86 processors. *Comput. Sci.*, 30, 2 (May 2015), 219–227. doi:10.1007/s00450-014-0270-z. http://dx.doi.org/10.1007/s00450-014-0270-z. (cited on page 67)

Sites, D., 2015. Data Center Computers: Modern Challenges in CPU Design. https://youtu.be/QBu2Ae8-8LM?t=3281. (cited on page 2)

Snavely, A.; Tullsen, D. M.; and Voelker, G., 2002. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 66–76. (cited on pages 22 and 36)

SPEC, 1999. *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edn. (cited on page 33)

SPEC, 2006. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation. http://www.spec.org/jbb2005. (cited on page 33)

Srivastava, A. and Eustace, A., 1994. Atom: A system for building customized program analysis tools. *SIGPLAN Not.*, 29, 6 (Jun. 1994), 196–205. doi:10.1145/773473.178260. http://doi.acm.org/10.1145/773473.178260. (cited on page 16)

Stevens, A., 2013. Introduction to AMBA 4 ACE^{TM} and big.LITTLE^{TM} Processing Technology. (cited on pages 22 and 36)

Strong, B., Sr, 2014. Debug and Fine-grain Profiling with Intel Processor Trace. In *Intel IDF14, San Francisco*. (cited on pages 10, 16, 47, and 49)

Wallace, S. and Hazelwood, K., 2007. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization*, 209–220. (cited on page 49)

Wamhoff, J.-T.; Diestelhorst, S.; Fetzer, C.; Marlier, P.; Felber, P.; and Dice, D., 2014. The TURBO diaries: Application-controlled frequency scaling explained. In *USENIX Annual Technical Conference (USENIX ATC)*, 193–204. (cited on page 100)

What is Kubernetes?, 2019. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (cited on page 73)

Wikipedia, 2016. Wikipedia:database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed January 2016. (cited on page 93)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2015a. Computer performance microscopy with shim. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15 (Portland, Oregon, 2015), 170–184. ACM, New York, NY, USA. doi:10.1145/2749469.2750401. http://doi.acm.org/10.1145/2749469.2750401. (cited on pages 19, 77, 88, and 93)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2015b. Shim open source implementation. https://github.com/ShimProfiler/SHIM. (cited on page 22)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2016a. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 309–322. USENIX Association, Denver, CO. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang. (cited on page 75)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2016b. Elfen scheduler open source implementation. https://github.com/elfenscheduler. (cited on page 78)

Yasin, A., 2014. A top-down method for performance analysis and counters architecture. In *IEEE Performance Analysis of Systems and Software (ISPASS)*, 35–44. (cited on pages 47 and 49)

Yi, J.; Maghoul, F.; and Pedersen, J., 2008. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, 257–266. (cited on page 75)

Zhang, X.; Wang, Z.; Gloy, N.; Chen, J. B.; and Smith, M. D., 1997. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97 (Saint Malo, France, 1997), 15–26. ACM, New York, NY, USA. doi:10.1145/268998.266640. http://doi.acm.org/10.1145/268998.266640. (cited on pages 16 and 47)

Zhang, Y.; Laurenzano, M. A.; Mars, J.; and Tang, L., 2014. SMiTe: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 406–418. (cited on pages 76, 79, and 100)

Zhang, Y.; Meisner, D.; Mars, J.; and Tang, L., 2016. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, 456–468. doi:10.1109/ISCA.2016.47. https://doi.org/10.1109/ISCA.2016.47. (cited on page 70)

Zhao, Q.; Cutcutache, I.; and Wong, W.-F., 2008. PiPA: Pipelined profiling and analysis on multi-core systems. In *International Symposium on Code Generation and Optimization*, 185–194. Boston, MA. (cited on pages 20 and 49)