Operating System Techniques for Reducing Processor State Pollution

by

Livio Soares

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Livio Soares

Abstract

Operating System Techniques for Reducing Processor State Pollution

Livio Soares
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2012

Application performance on modern processors has become increasingly dictated by the use of on-chip structures, such as caches and look-aside buffers. The hierarchical (multi-leveled) design of processor structures, the ubiquity of multicore processor architectures, as well as the increasing relative cost of accessing memory have all contributed to this condition. Our thesis is that operating systems should provide services and mechanisms for applications to more efficiently utilize on-chip processor structures. As such, this dissertation demonstrates how the operating system can improve processor efficiency of applications through specific techniques.

Two operating system services are investigated: (1) improving secondary and last-level cache utilization through a run-time cache filtering technique, and (2) improving the processor efficiency of system intensive applications through a new exception-less system call mechanism. With the first mechanism, we introduce the concept of a software pollute buffer and show that it can be used effectively at run-time, with assistance from commodity hardware performance counters, to reduce pollution of secondary on-chip caches.

In the second mechanism, we are able to decouple application and operating system execution, showing the benefits of the reduced interference in various processor components such as the first level instruction and data caches, second level caches and branch predictor. We show that exception-less system calls are particularly effective on modern multicore processors. We explore two ways for applications to use exception-less system calls. The first way, which is completely transparent to the application, uses multi-threading to hide asynchronous communication between the operating system kernel and the application. In the second way, we propose that applications can directly use the exception-less system call interface by designing programs that follow an event-driven architecture.

Acknowledgements

This is the part of the dissertation where grateful PhD students publicly recognize that the contents of this document was not a one person endeavor, but an effort made possible by a team of collaborators and supporters. However, the PhD is as much about a *path*, and a distinct chapter in our lives, as it is about the research produced. So these acknowledgments represent my humble attempt to show my gratitude to those who have inspired, mentored, and helped me along this path.

My path begins, naturally, with my family: Cássia, José and João. I was very fortunate to have been raised in a house where learning, curiosity and critical thinking were fostered. I feel grateful to have inherited unique values from each of my family members. It is clear to me now that these values were foundational in my decision to pursue a career path that I found meaningful and fulfilling.

My transition from an undergraduate to a graduate student was far from certain. In fact, I believe I could have easily chosen a different path if not for my master's advisor, Dilma da Silva. After all, I had not been an exemplary undergraduate student, and my interests were largely unaligned with the research of my department's faculty. Dilma played what is likely the most fundamental role in getting me into graduate school — she encouraged and allowed me to start a master's degree in an area I found exciting. She opened my eyes to the world of systems research and invited me to collaborate with her on her ongoing work at IBM Research. I'm very grateful for her selfless dedication to nurturing me in my early attempts at research.

Through Dilma I came to play a small part in the K42 research project, which has had an enormous influence on my research. I met wonderful researchers during my time with K42 — Jonathan Appavoo, Marc Auslander, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Volkmar Uhlig, Amos Waterland, Robert Wisniewski, and Jimi Xenidis — they were all a blast to work with! In particular, two of the researchers had a significant impact in my life and have served as role-models: Orran Krieger and Jonathan Appavoo. I thank them both for the support they demonstrated in my early stages and in the guidance in choosing a PhD program. Collaborating with them, whether coding at the break of dawn or discussing about the fundamentals of operating systems concepts, was both thrilling and formative. They have been great mentors to me throughout my PhD process; I couldn't have imagined a better way to be introduced to my field of research.

And through Orran and Jonathan, I was given the opportunity to come to the University of Toronto, under the guidance of Michael Stumm. My 7 years in Toronto were much more interesting than I imagined possible. My fellow colleagues in LP-372 made me feel at home right away: Reza Azimi, Adam Czajkowski, Alex Depoutovitch, Raymond Fingas, Gokul Soundararajan, Adrian Tam and David Tam. I thank them for the endless entertaining discussions. Special thanks to my closest collaborators at Toronto, Reza Azimi and David Tam. They were both patient with my naive enthusiasm, and they both taught me about the pains and joys of publishing research in academia. David has shown that consistent work and dedication can lead to surprising results. In addition, the work described in Chapter 3 was developed as an "offspring" to his own work on page-coloring. Without his collaboration, that work would have not existed. Reza showed me how to go from a rough idea of a research project to executing the research. More importantly, he was responsible for our group's focus and expertise in hardware performance counters. His initial effort in exploring hardware performance counters has greatly contributed to my work and was instrumental to the intuitions that led to my doctoral research.

Before choosing to join the University of Toronto, I came to visit Toronto with the hope that it would help me decide where to go for my PhD. Part of the visit involved meeting my future advisor, Michael Stumm. This meeting was very valuable to me. Although I did not know why exactly, I did feel that my experience working with Michael would be more enriching than elsewhere. I am grateful Michael picked out my PhD application from the graduate administrator's

trash can (legend has it). I also appreciate that when my personal life was in shambles, he was very supportive and generous towards me. I believe that my intuition that lead me to come to Toronto was spot-on. Michael has a talent of looking at the world with a slight fringe bias, while being able to articulate his ideas clearly and simply. For this alone, our weekly meetings were something I would look forward to, and will surely miss. I feel fortunate to have learned a thing or two from those meetings. From the technical side, his insistence on separating fundamental concepts from incidental side-effects is particularly important in our field. I believe it may be one of the tricks to asking great, yet simple questions — which can be a surprisingly valuable asset in research. From a less technical perspective, he has patiently reinforced to all of his students the value of communication, both in written and oral forms. Finally, his mentorship style, which I can at best describe as elegant and subtle, has been tremendously valuable throughout this entire process.

I thank the members of my thesis committee for their help and outside perspective. Ashvin Goel has always been a joy to discuss and brainstorm about systems research. During reading group discussions, it was fun to see his breadth of interests and enthusiasm for the papers we discussed. He was also particularly instrumental for motivating the work presented Chapter 6 of this dissertation. I thank Greg Steffan for his great course in parallel computer architecture. It was my first computer architecture course, but I felt that I needed a better background on computer architecture as an operating system researcher (little did I know computer architecture would play such a large role in my dissertation). Collaborating with Angela Demke-Brown was always a pleasure. Not only because her easygoing demeanor has made me feel comfortable interacting with her, but her comments are consistently thoughtful and helpful. Finally, I thank my external, Todd Mowry, for his time in getting to know my work, and his kind and enthusiastic evaluation of the dissertation work.

During my PhD, I was fortunate to collaborate with industrial research labs and broaden the scope of my research experience. I thank the research groups at IBM Research and Intel for these valuable opportunities. At IBM Research for a second time, with Dilma da Silva, Bryan Rosenburg and Maria Butrico, I learned quite a bit on virtualization and building minimalistic operating systems. At Intel I worked with an interesting set of researchers who knew much more about hardware than I did: Mani Azimi, Naveen Cherukuri, Ching-Tsun Chou, Donglai Dai, Akhilesh Kumar, Partha Kundu, Dongkook Park, Seungjoon Park, Roy Saharoy, Anahita Shayesteh, Hariharan L. Thantry, and Aniruddha S. Vaidya. They gave me a fascinating glimpse on prototyping hardware on a complex industrial project.

On my path to complete my graduate studies, I started in Sao Paulo (Brazil), passing through Yorktown Heights (NY), Toronto (Canada), Mount Kisco (NY), Mountain View (CA), and Croton-on-Hudson (NY). In many ways, it was an exciting adventure. But the most impactful part of the adventure, and frankly, the most unexpected, was meeting my best friend and life partner, Ioana. It's hard to imagine going through this adventure without her by my side. I admire many things in Ioana. But specific to my work, she has been a true mentor to me. She embodies a rare combination of uncommon intelligence and uncommon ability to care. She has shown tremendous care and persistence in helping me when I felt stuck. Despite the fact that I've tested her patience, and drained her energy from time to time with my frustrations with work, general indecisions, stubbornness with getting just the right shade of orange for my slides, and carelessness in discussing work when we should have been trying to relax, she never gave up on helping me. Her energetic personality, witty intelligence, and endless curiosity for the things she finds truly meaningful were always inspiring to me. Ultimately, I believe that she has inspired me to make this work better than "good enough". I am certain that through her example I have allowed and pushed myself to dream bigger dreams. I'm very excited to be able to share the next chapter of the adventure with Ioana.

Contents

1 Introduction and Motivation **1**
1.1 Thesis . . . 3
1.2 Dissertation Outline . . . 5
1.2.1 Software Pollute Buffer . . . 5
1.2.2 Exception-less System Calls . . . 6
1.2.3 Exception-less Threads . . . 8
1.2.4 Exception-less Event-driven Programming . . . 9
1.3 Summary of Research Contributions . . . 10

2 Background and Related Work **12**
2.1 Computer Hardware . . . 12
2.1.1 Fast Processor; Dense (not so fast) Memory . . . 12
2.1.2 Multicore Processors . . . 15
2.1.3 Processor Caches, Buffers and Tables . . . 16
2.1.4 Prefetching and Replacement Algorithms . . . 17
2.1.5 Prefetching . . . 18
2.1.6 Replacement . . . 19
2.1.7 Eliminating Mapping Conflict Misses in Direct-Mapped Structures . . . 20
2.1.8 Cache Bypassing . . . 20
2.2 Computer System Software . . . 22
2.2.1 Virtualization and OS Abstractions . . . 22
2.2.2 Support for Parallelism . . . 23
2.2.3 I/O Concurrency: Threads and Events . . . 24
2.2.4 Locality of Execution and Software Optimizations for Processor Caches . . . 28
2.2.5 Page Coloring and Software Cache Partitioning . . . 29
2.2.6 Operating System Interference . . . 30
2.2.7 Optimizing Software Communication: IPC and System Calls . . . 32

3 Software Pollute Buffer **34**
3.1 Introduction . . . 35
3.2 Background . . . 36
3.2.1 Software Cache Partitioning . . . 36
3.2.2 Hardware Performance Counters . . . 38
3.3 Address-Space Cache Characterization . . . 39
3.3.1 Exploiting Hardware Performance Counters . . . 39
3.3.2 Empirical Simulation-based Validation . . . 40
3.3.3 Page-Level Cache Behavior . . . 44
Classifying Pollution . . . 44
Case Study: *art* . . . 46
Prefetching Interference . . . 47

3.4 Software-Based Cache Pollute Buffer . . . 48
3.4.1 Kernel Page Allocator . . . 49
3.5 Run-Time OS Cache-Filtering Service . . . 50
3.5.1 Online Profiling . . . 50
3.5.2 Dynamic Page-Level Cache Filtering . . . 51
3.6 Evaluation . . . 52
3.6.1 Overhead . . . 54
3.6.2 Performance Results . . . 55
3.6.3 Case study: *art* . . . 57
3.6.4 Case study: *swim* . . . 57
3.7 Discussion . . . 59
3.7.1 Limitations . . . 59
3.7.2 Stall-rate oriented profiling . . . 60
3.7.3 Software managed/assisted processor caches . . . 60
3.8 Summary . . . 61

4 Exception-less System Calls **63**
4.1 Introduction . . . 63
4.2 The (Real) Costs of System Calls . . . 65
4.2.1 Mode Switch Cost . . . 65
4.2.2 System Call Footprint . . . 67
4.2.3 System Call Impact on User IPC . . . 68
4.2.4 Mode Switching Cost on Kernel IPC . . . 70
4.2.5 Significance of system call interference experiments . . . 70
4.3 Exception-Less System Calls . . . 71
4.3.1 Exception-Less Syscall Interface . . . 72
4.3.2 Syscall Pages . . . 72
4.3.3 Decoupling Execution from Invocation . . . 74
4.4 Implementation – FlexSC . . . 74
4.4.1 `flexsc_register()` . . . 75
4.4.2 `flexsc_wait()` . . . 75
4.4.3 Syscall Page Allocation . . . 76
4.4.4 Syscall Threads . . . 77
4.4.5 FlexSC Syscall Thread Scheduler . . . 78
4.5 Summary . . . 80

5 Exception-Less Threads **82**
5.1 FlexSC-Threads Overview . . . 82
5.2 Multi-Processor Support . . . 85
5.2.1 Per core data structures and synchronization . . . 85
5.2.2 Thread migration . . . 86
5.2.3 Syscall pages . . . 88
5.3 Limitations . . . 88
5.4 Experimental Evaluation . . . 89
5.4.1 Overhead . . . 90
5.4.2 Apache . . . 91
5.4.3 MySQL . . . 96
5.4.4 BIND . . . 100
5.4.5 Sensitivity Analysis . . . 104
5.5 Discussion . . . 105
5.5.1 Increase of user-mode TLB misses . . . 105

5.5.2 Latency . . . 106
5.6 Summary . . . 106

6 Event-Driven Exception-Less Programming 108
6.1 Introduction . . . 108
6.2 *Libflexsc*: Asynchronous system call and notification library . . . 111
6.2.1 Example server . . . 112
6.2.2 Cancellation support . . . 113
6.3 Exception-Less Memcached and nginx . . . 114
6.3.1 Memcached - Memory Object Cache . . . 114
6.3.2 nginx Web Server . . . 115
6.4 Experimental Evaluation . . . 115
6.4.1 Memcached . . . 116
6.4.2 nginx . . . 119
ApacheBench . . . 119
httperf . . . 122
6.5 Discussion: Scaling the Number of Concurrent System Calls . . . 124
6.6 Summary . . . 125

7 Concluding Remarks 126
7.1 Lessons Learned . . . 128
7.1.1 Difficulty of assessing and predicting performance . . . 128
7.1.2 Run-time use of hardware performance counters . . . 129
7.1.3 Interference of prefetching on caching . . . 130
7.1.4 Cost of synchronization . . . 131
7.2 Future Work . . . 131
7.2.1 *Hardware Introspection* through advanced hardware performance counters . . . 132
7.2.2 Hardware support for *event-based code injection* . . . 134
7.2.3 Exposing *software buffer* to language or compiler . . . 134
7.2.4 Software assisted cache management . . . 135
7.2.5 Lightweight inter-core notification and communication . . . 136
7.2.6 Interference aware profiling . . . 137
7.2.7 Execution slicing: pipelining execution on multicores . . . 138

Bibliography 139

List of Tables

2.1 Cache hierarchy characteristics for x86 based processors. 16

3.1 Cache parameters using in simulation-based experiments 41
3.2 Characteristics of the 2.3GHz PowerPC 970FX 53
3.3 SPEC CPU 2000 Benchmark characteristics 53
3.4 Classification of pollute pages 57

4.1 Micro-benchmark system call overhead 66
4.2 System call footprints 67
4.3 Number of instructions between syscalls. 69

5.1 Core i7 processor characteristics 90
5.2 Micro-architectural breakdown for Apache 94
5.3 Micro-architectural breakdown for MySQL 98
5.4 Micro-architectural breakdown for BIND 103

6.1 Number of instructions per system call for memcached and nginx 110
6.2 Comparison of invocation and execution models for user-kernel communication . . 111
6.3 Code level statistics on porting nginx and memcached to libflexsc 114
6.4 Micro-architectural breakdown for Memcached 118
6.5 Micro-architectural breakdown for nginx 123

List of Figures

1.1 Simple illustration of the *software pollute buffer* . . . 6
1.2 Synchronous versus exception-less system calls . . . 7
1.3 Component-level overview of FlexSC . . . 7
1.4 FlexSC-Threads illustration . . . 9

2.1 Historic trend of number of transistors in processor chips . . . 13
2.2 Historic trend of memory capacity . . . 13
2.3 Performance gap between processor and main memory (DRAM) . . . 14
2.4 Cache indexing bit-fields. . . . 29

3.1 Cache indexing bit-fields. . . . 36
3.2 Example of L2 cache partitioning through page coloring . . . 37
3.3 Simulation-based validation of AMMP . . . 42
3.4 Simulation-based validation of ART . . . 42
3.5 Simulation-based validation of MGRID . . . 43
3.6 Simulation-based validation of SWIM . . . 43
3.7 Page-level L2 cache miss rate characterization. . . . 45
3.8 Page-level L2 cache miss characterization for *art* . . . 46
3.9 Page-level L2 cache miss rate characterization for *wupwise*, with and without prefetching . . . 47
3.10 Software Pollute Buffer . . . 49
3.11 Overhead sensitivity of monitoring *art*. . . . 51
3.12 Run-time overhead breakdown of ROCS . . . 54
3.13 Performance improvement of ROCS over default Linux . . . 56
3.14 MPKI reduction with ROCS over a default Linux . . . 56
3.15 Effect of cache filtering on *art* . . . 58

4.1 User-mode IPC recovery after system call . . . 64
4.2 Impact of `pwrite` on Xalan . . . 68
4.3 Impact of `pwrite` on SPEC JBB 2005 . . . 69
4.4 Impact of mode switching on kernel IPC . . . 70
4.5 Synchronous versus exception-less system calls . . . 71
4.6 Example system call invocation, showing syscall page . . . 73
4.7 Overview of FlexSC . . . 75
4.8 Example of FlexSC on multicore . . . 79

5.1 FlexSC-Threads illustration . . . 83
5.2 FlexSC-Threads on multicore . . . 86
5.3 Exception-less syscall overhead on single core . . . 90
5.4 Exception-less syscall overhead on remote core . . . 91
5.5 Apache throughput comparison . . . 92

5.6 Micro-architectural breakdown for Apache 94
5.7 Kernel, user and idle times for Apache 95
5.8 Apache latency comparison 95
5.9 Kernel, user and idle times for MySQL 96
5.10 MySQL throughput comparison 97
5.11 Micro-architectural breakdown for MySQL 98
5.12 MySQL latency comparison 100
5.13 Kernel, user and idle times for BIND 101
5.14 BIND throughput comparison 101
5.15 Micro-architectural breakdown for BIND 103
5.16 BIND latency comparison 104
5.17 FlexSC sensitivity to the number of syscall entries 105

6.1 Example of network server using libflexsc 112
6.2 Memcached throughput comparison 117
6.3 Micro-architectural breakdown for Memcached 118
6.4 nginx throughput comparison 120
6.5 nginx latency comparison with ApacheBench 120
6.6 nginx performance with the *httperf* workload 121
6.7 Micro-architecture breakdown for nginx 123

Chapter 1

Introduction and Motivation

Computer systems, largely fueled by exponential increases in computing performance, has and continues to change our society in profound ways. Increasingly, we rely on computing as a fundamental infrastructure – as fundamental as electricity became with the Industrial Revolution. The speed and ubiquity of computing infrastructure has enabled a previously unimaginable number of uses for computers and processors. We believe that the ability to offer faster computing, at lower costs, directly and indirectly benefits our society and most sectors of the economy.

A key component contributing to the improvements in computing performance is the computer processor. The ability to shrink transistor feature sizes over decades has allowed computer processors to offer drastic performance improvements. These improvements stem from both an increase in the number of available transistors in an integrated processor, which has enabled the construction of more complex logic to support computation (e.g., superscalar and out-of-order execution), as well as an increase in the switching speed of transistors, which has allowed circuits to be clocked at higher frequencies.

At this point in history, the growth of single processor performance is decreasing, and is not expected to improve in next few decades [80]. Processors have reached physical and engineering limits that have prevented the same speed increases of previous decades. These limits stem from heat and thermal issues that limit improvements in transistor switching speed, as well as digital logic design and automation that make it cost ineffective to architect and produce single processors that integrate several billion transistors.

Given these technological trends, chip manufactures have been forced to redesign and re-engineer processing chips so that they continue to provide significant performance improvements from one generation to the next. The main strategy used by mainstream chip manufactures has been to adopt multicores, where the abundance of transistors available on a single chip are used to implement multiple independent processors.

Another principal component of modern computers, main memory, has also evolved significantly over the past few decades. In particular, as a response to the constant demand for larger memory sizes, the DRAM industry has focused on increasing the density of memory devices and

reducing the cost per bit. The exponential growth in memory capacities has had an enormous impact in the applicability of computers to new domains — allowing the deployment of an increasing class of computations and applications.

Partly because of the focus on capacity and price, the speed of memory has not increased at the same pace as that of processors. In fact, while accessing a word of memory took roughly the same amount of time as executing an arithmetic operation in the early 1980s, today it is possible to execute several hundred arithmetic operations in the time span of a memory access. This *memory performance gap* is the main reason on-chip processor caches have been incorporated into every mainstream processor, with the goal of reducing the average latency to memory.

The rise of multicore processors and the memory performance gap has meant that *communication*, whether between processor and memory or between multiple processors, has played a central role in the performance of computers. In the context of modern processors which rely heavily on hierarchical multi-leveled caches, aggressive prefetchers, and coherency between multiple cores, communication occurs mostly implicitly when programs simply access data.

In order to improve performance impacted by long communication latencies, several techniques have been proposed and studied, particularly promoting the “principle of locality” [60, 61, 62]. To date, the majority of these techniques have centered around optimizations in the underlying hardware, as well as improving the quality of machine code either through manual or compiler-based optimizations. While these techniques have been successful in improving the performance of applications by reducing communication or hiding communication overheads, there are still workloads that do not make optimal use of on-chip structures and consequently are negatively impacted by communication latencies.

In this dissertation, we contend that the operating system can play a unique role in improving the performance of applications. We focus on *processor state pollution* that occurs when items of a processor component (such as cache lines or TLB entries) that are to be accessed in the near future are displaced by items that are not re-accessed. As a consequence, displaced items must be re-fetched when subsequently accessed, increasing the amount of implicit communication which can negatively impact performance and execution efficiency.

Specifically, we explore addressing two types of execution interference that result in processor state pollution using operating system level techniques. The first interference we address is *intra-application* interference of secondary caches (i.e., the large caches above the first level of cache). We observe that in applications that make poor use of the processor’s secondary caches, there are regions of the address space, typically larger than a page in length, that uniformly exhibit little or no reuse. During run-time, the data of these regions are placed in the cache when accessed, potentially evicting the other more useful data items. Eliminating the intra-application interference in secondary caches improves performance by allowing reusable data items to be fetched from the cache hierarchy more often, thus reducing the average cost of accessing memory.

The second type of interference we explore is the one between the application and the operat-

ing system kernel. We find that when applications make heavy use of operating system services, as is often the case with server-type applications, there is fine-grain multiplexing of application and operating system execution. We show that the fine grain multiplexing of these two execution modes does not produce localized accesses to processor structures as the differing working sets compete for space in processor structures.

1.1 Thesis

Our thesis is that operating systems should provide services and mechanisms to allow applications to more efficiently utilize on-chip processor structures. To this end, this dissertation introduces and explores two novel techniques that improve application performance by eliminating processor state pollution. First, we describe an operating system cache filtering mechanism, implemented in software with the assistance of hardware performance counters, with the goal of improving the effectiveness of secondary on-chip caches.

Second, we describe a new operating system mechanism, called *exception-less system call*, that improves locality of execution of operating system intensive workloads. Exception-less system calls allow execution of applications to be decoupled from operating system execution; this decoupling is exploited to schedule execution such that interference between application and operating system is reduced. In addition, this mechanism enables innovative execution on multicores that makes more efficient use of per-core on-chip structures by allowing cores to be dynamically specialized with operating system or application execution.

Traditionally, the responsibilities for optimizing use of on-chip components such as caches and communication buses have fallen to the hardware itself, manual machine code transformations, or compiler based optimizations. However, in this thesis we argue that the operating system is uniquely placed in the compute stack and should be a natural layer for implementing certain mechanisms, such as those that reduce processor state pollution. In particular, we argue that there are mechanisms that are not amenable to being implemented in hardware or through machine code transformations, and are only suitable to be deployed within the run-time and operating system[1]. The characteristics unique to operating systems which are explored in this dissertation are:

1. Ability to monitor and react to run-time execution. While some locality optimizations can be deployed statically, and can therefore be introduced manually in the program or through a compiler transformation, other optimizations must respond to run-time behavior or execution platform. Reasons for this include: differences in workload inputs often result in different execution behavior; the variety of cache sizes and geometries of different machines ac-

[1] Throughout this dissertation, we refer to the run-time environment and libraries and the operating system kernel as a single layer in the software stack, namely, the *operating system* layer. Even though it may be that run-time libraries execute in user-mode and the operating system kernel executes in a privileged, kernel-mode, both components constitute essential parts of modern operating systems.

commodate different working set sizes; differences in latencies may result in executions that exhibit distinct performance bottlenecks; concurrently running applications can also make a significant impact in the availability of resources such as on-chip and off-chip buses and shared on-chip caches.

At the operating system level, it is possible to monitor and track different run-time characteristics of programs. In fact, it already does so in certain cases such as page-level access pattern for virtual memory and software TLB management, as well as file-system prefetching. Therefore, extending the existing operating system infrastructure to monitor more features of applications should pose a relatively low barrier towards adoption.

2. Less complex, and cheaper, than hardware to deploy and modify. Unfortunately, designing and verifying new extensions to general purpose processors is still both economically expensive and implies a high turn-around time. This requires companies to focus on features that are considered economically beneficial, primarily for common usage, which in turn limits the types of enhancements that are adopted.

   Operating systems, and software in general, have lower costs associated with development, and most importantly lower turn-around times. This makes it viable to incorporate specialized optimizations in the operating system that may benefit a smaller fraction of applications.

   Finally, certain optimizations are prohibitively expensive in terms of storage requirements to be implemented completely in hardware. For example, if an optimization must collect over several megabytes of run-time information, then dedicating hardware for such an optimization may be prohibitively expensive. At the operating system level, however, because metadata can be maintained in main memory, dedicating megabytes of memory to an optimization is a manageable cost since it represents a small fraction of overall main memory capacity.

3. Access to both the semantics of software and low-level hardware information. Due to the fundamental responsibility of an operating system, namely to interface applications with the underlying hardware, the operating system has the ability to monitor both application execution and, through hardware performance counters, the effect execution has on most processor components.

   For example, abstractions such as application data structure, function call stack, virtual address space, threading, shared memory and inter-process communication are accessible in the operating system; at the processor level, however, inferring this type of information is difficult and potentially expensive. This semantic gap that exists between application and processor means that some optimizations are intractable to implement at the hardware level. We believe that the run-time and operating system layer is the most adequate layer of the computer stack to fill the software-hardware semantic gap.

1.2 Dissertation Outline

In Chapter 2 we provide a brief summary of trends in the evolution of computer systems, focusing on aspects that relate to and further motivate the work described in this dissertation. Our *software pollute buffer* technique is presented in Chapter 3. The *exception-less system call* mechanism is presented in Chapter 4, along with experiments that show how the interrupt-based system call mechanism that is widely used today has a negative impact on processor structures. In the following two chapters, Chapter 5 and 6, we explore two ways of using exception-less system calls. First, we describe a threading based solution, that requires no changes to existing multi-threaded programs. Secondly, we explore programs that directly interface with exception-less system calls with an event-driven programming library to assist programmers. Chapter 7 concludes the dissertation, discussing some of lessons we learned and future research directions.

In the remainder of this chapter, we provide a brief overview of the techniques described in the subsequent chapters.

1.2.1 Software Pollute Buffer

It is well recognized that the least recently used (LRU) replacement algorithm can be ineffective for applications with large working sets or non-localized memory access patterns [85, 97, 98, 154, 209]. Specifically, in secondary processor caches, LRU can cause cache pollution by inserting non-reuseable elements into the cache while evicting reusable ones. Despite advances in compiler optimizations and hardware prefetchers, we, along with other researchers [93, 94, 154], observe that there are classes of workloads that exhibit poor use of secondary on-chip caches. We argue that the low efficiency of secondary caches is partly due to *intra-application interference* that causes pollution in these caches.

In Chapter 3, we explore an operating system technique to improve the performance of applications that exhibit high miss rates in secondary processor caches. A principal insight behind our technique is that, for certain applications, access patterns are distinct for different regions of an application's address space. In the case that one of the regions exhibits LRU unfriendly access patterns, there is potential for intra-application interference in the cache hierarchy. We establish two properties of memory intensive workloads: (1) applications exhibit large-spanning virtual memory regions, each exhibiting a uniform memory access pattern, and (2) at least one of the regions does not temporally reuse cache lines.

We propose addressing secondary-level cache pollution resulting from intra-application interference through a dynamic operating system mechanism, called **ROCS**, requiring no change to underlying hardware and no change to applications. ROCS exploits hardware performance counters on a commodity processor to characterize application cache behavior at run-time. Using this online profiling, cache unfriendly pages are dynamically mapped to a *pollute buffer* in the cache, eliminating competition between reusable and non-reusable cache lines. The operating system im-

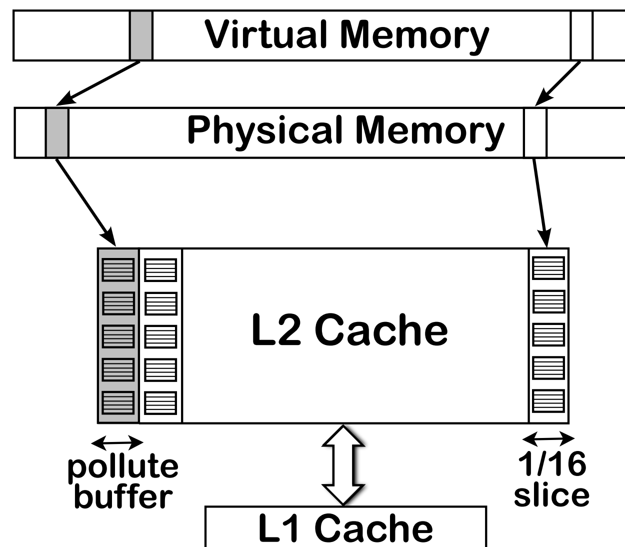


Figure 1.1: Representation of a *software pollute buffer*. The software pollute buffer is implemented by dedicating a partition of a secondary level cache to host lines from pages that cause cache pollution. To implement the pollute buffer, we exploit a well-known operating system technique called page coloring. At a high level, the operating system can map application virtual pages (top box) to a selected set of physical pages. These physical pages are selected based on their address so that, according to the indexing function of the secondary cache, the content of the pages will occupy a fixed, and small, partition of the cache.

plements the pollute buffer through a page-coloring based technique [118, 125, 202], by dedicating a small slice of the last-level cache to store non-reusable pages, as depicted in Figure 1.1. Measurements show that ROCS, implemented in the Linux 2.6.24 kernel and running on a 2.3GHz PowerPC 970FX, improves performance of memory-intensive SPEC CPU 2000 and NAS benchmarks by up to 34%, and 16% on average.

1.2.2 Exception-less System Calls

For the past 30+ years, system calls have been the *de facto* interface used by applications to request services from the operating system kernel. System calls have almost universally been implemented as a *synchronous* mechanism, where a special processor instruction is used to yield user-space execution to the kernel, typically through an interrupt or processor exception. Certain classes of applications, such as server-type applications, make heavy of operating system services. During execution, these applications make calls into the operating system as frequently as once for every few thousands of instructions.

In Chapter 4, we evaluate the performance impact of traditional synchronous system calls on system intensive workloads. We show that synchronous system calls negatively affect performance in a significant way, primarily because of pipeline flushing and pollution of key processor structures (e.g., TLB, data and instruction caches, etc.). The pollution observed in various processor structures stems from interference between the execution of application and the operating system

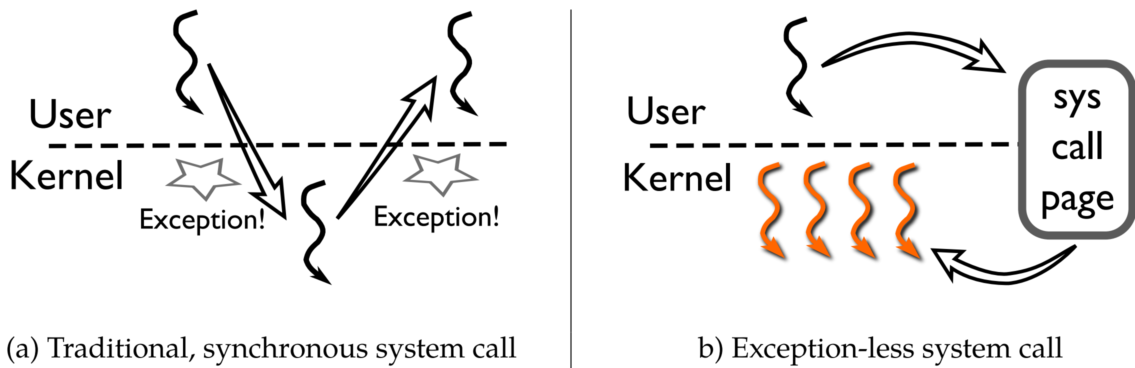


Figure 1.2: Illustration of synchronous (a) and exception-less (b) system call invocation. The wavy lines are representation of threads of execution (user or kernel). The left diagram illustrates the sequential nature of exception-based system calls. When an application thread makes a system call, it uses a special instruction that generates a processor interrupt or exception. The processor immediately transfers control to the operating system kernel, where the call is executed synchronously. After which, the kernel returns control to the application thread, which is done through an exception-based mechanism similar to the system call. The right diagram, on the other hand, depicts exception-less user and kernel communication. Messages are exchanged asynchronously through a portion of shared memory, which we call *syscall page*, by simply reading from and writing to it.

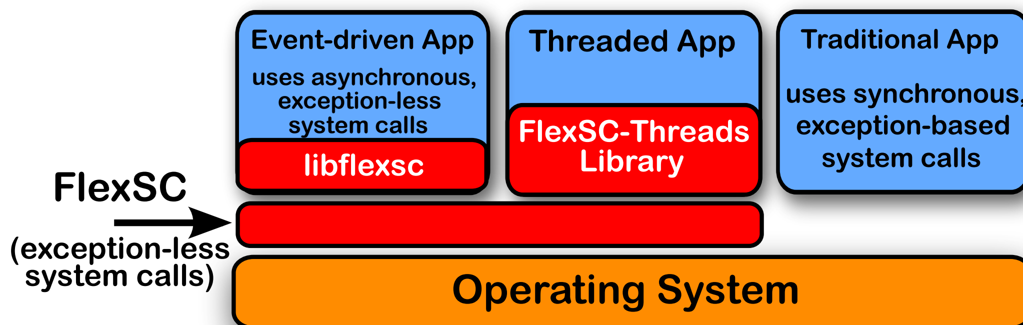


Figure 1.3: Component-level overview of FlexSC. The implementation of operating system services, representative by the bottom box, are not altered by our FlexSC system. As a consequence, legacy applications that use exception-based system call mechanism continue to work unaltered. We introduce a new operating system mechanism, exception-less system calls (FlexSC), that can be used by applications to asynchronously request operating system services. We also introduce two new libraries, FlexSC-Threads which is intended to support legacy thread based programs in a transparent way, and *libflexsc* which supports event-driven applications that directly make use of FlexSC.

kernel.

We propose a new mechanism for applications to request services from the operating system kernel: *exception-less system calls*. While the traditional system call mechanism requires a processor exception to synchronously communicate with the kernel, as well as to reply to the application, exception-less system calls, instead, rely on messages that are exchanged completely asynchronously through memory. Figure 1.2 depicts the interaction between user and kernel modes with a traditional synchronous system call mechanism and with the proposed exception-less mechanism. In Chapter 4, we describe an implementation of exception-less system calls, which we call *FlexSC* (for flexible system call scheduling), within the Linux kernel. A high level overview of the components added to a traditional software stack is show in Figure 1.3.

Exception-less system calls improve processor efficiency by enabling flexibility in the scheduling of operating system work, which in turn can lead to significantly increased temporal and spacial locality of execution in both user and kernel space, thus reducing pollution effects on processor structures. Exception-less system calls are particularly effective on multicore processors as they allow the operating system to dynamically execute operating system and application code on separate cores which improves executoin locality. They primarily target highly parallel server applications, such as Web servers and database servers.

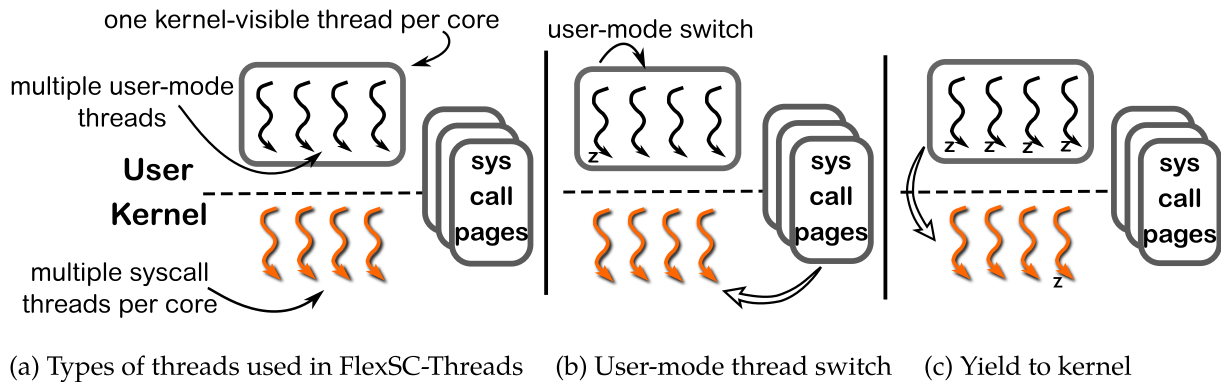
A main difference between synchronous and exception-less system calls, from the application's perspective, is the programmability of each interface. Because exception-less system calls are asynchronous, they can pose an onus on the programmer to operate with a more complex interface to the operating system. We address programmability aspects of exception-less system calls by describing a solution based on multi-threaded programming and another based on event-driven programming in the following two subsections.

1.2.3 Exception-less Threads

To benefit from asynchronous operating system execution, applications must execute useful work while waiting for operating system calls to complete. One way to do so is to rely on multi-threaded applications. With various independent threads within an application, it is possible to multiplex the execution of threads that are currently waiting for system calls to complete, with threads that are not waiting on operating system work.

In Chapter 5, we describe the design and implementation of a user-level threading library (FlexSC-Threads) that allows existing multi-threaded applications to transparently use exception-less system calls. FlexSC-Threads uses a simple M-on-N threading model (M user-mode threads executing on N kernel-visible threads). It relies on the ability to perform user-mode thread switches solely in user-space to transparently transform legacy synchronous calls into exception-less ones. Figure 1.4 depicts a simple example of how user-level threading is used in FlexSC-Threads along with the interaction with the exception-less system call mechanism.

By treating a system call as a blocking operation within the threading library, from the perspec-



(a) Types of threads used in FlexSC-Threads (b) User-mode thread switch (c) Yield to kernel

Figure 1.4: Three diagrams that describe the interaction between FlexSC-Threads and FlexSC. The left-most diagram (a) depicts the components of FlexSC-Threads pertaining to a single core. FlexSC-Threads uses a kernel-visible thread to multiplex the execution of multiple user-mode threads. Multiple syscall pages, and consequently syscall threads, are also allocated per kernel-visible thread. The middle diagram (b) depicts what happens after a user thread makes a system call. The user thread is blocked, and another thread from the ready queue is chosen to run; the thread switch occurs completely in user-mode. In the background, a syscall thread can begin executing the system call. The right-most diagram (c) depicts the scenario where all user-mode threads are waiting for system call requests; in this case FlexSC-Threads library synchronously yields to the kernel. Syscall threads can be woken to execute pending system calls.

tive of each thread the system call interface is maintained since the asynchronous implementation of the system call is hidden by the library. From the application's perspective, however, execution can proceed without waiting or switching into the kernel given that there are sufficient independent threads to schedule. The implementation of FlexSC-Threads is compliant with POSIX Threads, and binary compatible with NPTL [65], the default Linux thread library. As a result, Linux multi-threaded programs work with FlexSC-Threads "out of the box" without modification or recompilation.

The performance evaluation we present focuses on popular multi-threaded server applications. In particular, we show that our implementation of exception-less system, in conjunction with our specialized threading library, improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 79% while requiring no modifications to the applications.

1.2.4 Exception-less Event-driven Programming

To maximize performance, application writers may be willing to (re)write programs that directly use the exception-less system call interface. In fact, event-driven architecture is a popular software pattern that has traditionally relied on asynchronous operations, akin to exception-less system calls, for constructing scalable, high-performance server applications. Due to the popularity of event-driven architectures, operating systems have invested in efficiently supporting non-blocking and asynchronous I/O, as well as scalable event-based notification systems. To leverage the experience the software community has had with event-driven architectures, we explore exposing exception-less system calls in a way that is suitable for constructing event-driven applications.

In Chapter 6, we first show that the direct and indirect performance overheads associated with high frequency of system calls are present in the execution of event-driven server applications, even when using modern interfaces for asynchronous I/O and event notification. Subsequently, we propose the use of *exception-less system calls* as the main operating system mechanism to construct high-performance event-driven server applications. Exception-less system calls have four main advantages over traditional operating system support for event-driven programs: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor based, (2) support in the operating system kernel is non-intrusive as code changes are not required for each system call, (3) processor efficiency is increased since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multi-core execution for event-driven programs is easier, given that a single user-mode execution context can generate enough requests to keep multiple processors/cores busy with kernel execution.

We present *libflexsc*, an asynchronous system call and notification library suitable for building event-driven applications. Libflexsc makes use of exception-less system calls through our Linux kernel implementation, FlexSC. We describe the port of two popular event-driven servers, *memcached* and *nginx*, to libflexsc. We show that exception-less system calls increase the throughput of memcached by up to 35% and nginx by up to 120% as a result of improved processor efficiency.

1.3 Summary of Research Contributions

In this dissertation, we aim to provide compelling evidence that operating systems should provide services and mechanisms for applications to more efficiently utilize on-chip processor structures. To this end, we developed specific operating system techniques that reduce pollution of different processor components. We believe we have identified and targeted common sources of pollution that are found in existing classes of workloads and that have not been fully addressed by optimizations within other layers of the computer stack. Furthermore, we argue that the run-time and operating system are the natural layers to implement these optimizations.

The specific techniques introduced and evaluated in this dissertation are:

- **Software pollute buffer**. We develop an operating system cache filtering service, that is applied at run-time and improves the effectiveness of secondary processor caches. We identify intra-application interference as an important source of pollution in secondary on-chip caches. Leveraging commodity hardware performance units, we demonstrate how to generate application address space cache profiles at run-time with low overhead. The online profile is used to identify regions of memory or individual pages that cause pollution and do not benefit from caching. Finally, we show how page-coloring can be used to create a *software pollute buffer* in secondary caches to restrict the interference caused by the polluting regions of memory.

- **Exception-less system calls**. We develop a novel mechanism, called exception-less system call, that allows applications to request operating system services with low overhead and asynchronously schedule operating system work on multiple cores. We quantify the impact of system calls on the performance of system intensive workloads, showing that there are direct and indirect components to the overhead. We propose a new system call mechanism, exception-less system calls, that uses asynchronous communication through the memory hierarchy. An implementation of exception-less system calls, called FlexSC, is described within a commodity monolithic kernel (Linux), demonstrating the applicability of the mechanism to legacy kernel architectures.

- **Exception-less user-level threading**. We develop a new hybrid threading package, FlexSC-Threads, specifically tailored for use with exception-less system calls. The goal of the presented threading package is to translate legacy system calls to exception-less ones *transparently* to the application. We experimentally evaluate the performance advantages of exception-less execution on popular server applications, showing improved utilization of several processor components. In particular, our system improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 79% while requiring no modifications to the applications.

- **Exception-less event driven programming**. We explore exposing exception-less system calls directly to applications. To this end, we develop a library that supports the construction of event-driven applications that are tailored to request operating system services asynchronously. We show how to port existing event-driven applications to use our new mechanism. Finally, we identify various benefits of exception-less system calls over existing operating system support for event-driven programs. We show how the use of direct use of exception-less system calls can significantly improve the performance of two Internet servers, *memcached* and *nginx*. Our experiments demonstrate throughput improvements in *memcached* of up to 35% and *nginx* of up to 120%. As anticipated, experimental analysis shows that the performance improvements largely stem from increased efficiency in the use of the underlying processor when pollution is reduced.

Chapter 2

Background and Related Work

This chapter provides background on key aspects of computer systems that relate to the work presented in this dissertation. We divide this chapter into two major sections, one dedicated to computer hardware, and one dedicated to system software, with a focus on operating systems. In both sections, we highlight past and current trends, in part, to communicate aspects that motivated the work presented in subsequent chapters. We also describe previous research and highlight studies that have influenced our work.

2.1 Computer Hardware

Modern computer hardware, from an abstract point of view, is still based on the von Neumann architecture [142]. Today, computers consist of a central processing unit, largely based on digital logic, volatile memory, and persistent storage. Yet at the same time, computer hardware has undergone tremendous changes over the past 50 years as they have become *billions* of times more powerful. In this section, we describe trends that have enabled these transformations, as well as how, due to technological constraints, we can expect to observe more profound changes in the coming years. In addition, we describe some of the challenges that are current areas of research, focusing on performance of computers.

2.1.1 Fast Processor; Dense (not so fast) Memory

Since the introduction of the first integrated microprocessors in the early 1970s, mainstream computers have relied on digital devices as their basic building block. The widespread adoption of digital systems stems from the fact that they are easy to reprogram, and allow for accurate and deterministic operation. Underlying the digital systems used in computers, and serving as their foundation, is the *transistor* — a semiconductor device that acts as a switch in a digital circuit.

The commercial success of transistors has led the semiconductor industry to focus on transistor scaling, allowing more transistors to be packaged into integrated circuits without increasing, and often lowering, costs. Figures 2.2 and 2.1 show the effect of transistor scaling on two central

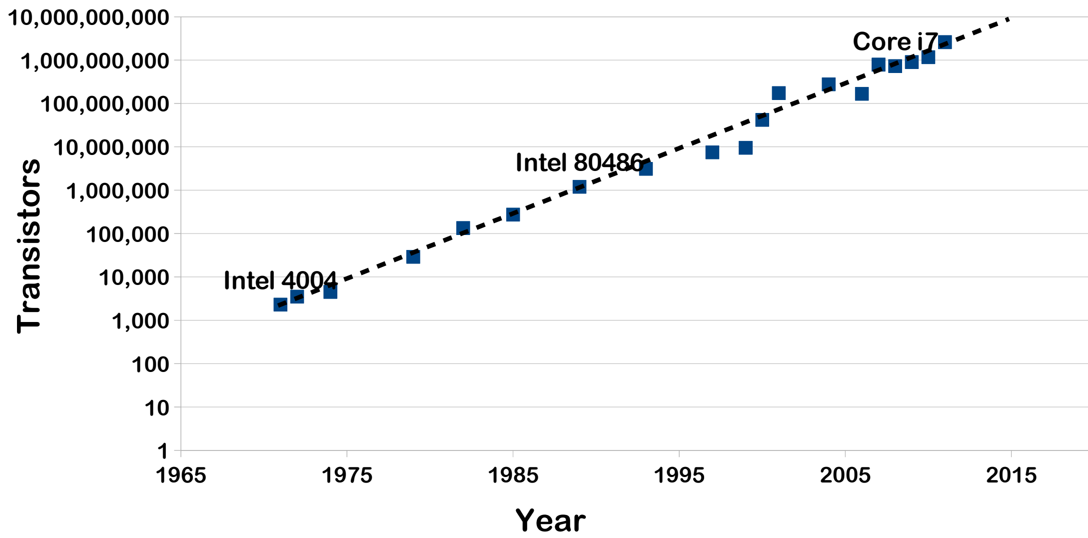


Figure 2.1: Historic trend of number of transistors in processor chips. Survey of high-end desktops and low-end servers. *Sources*: itrs.net, wikipedia.org, intel.com, amd.com and ibm.com

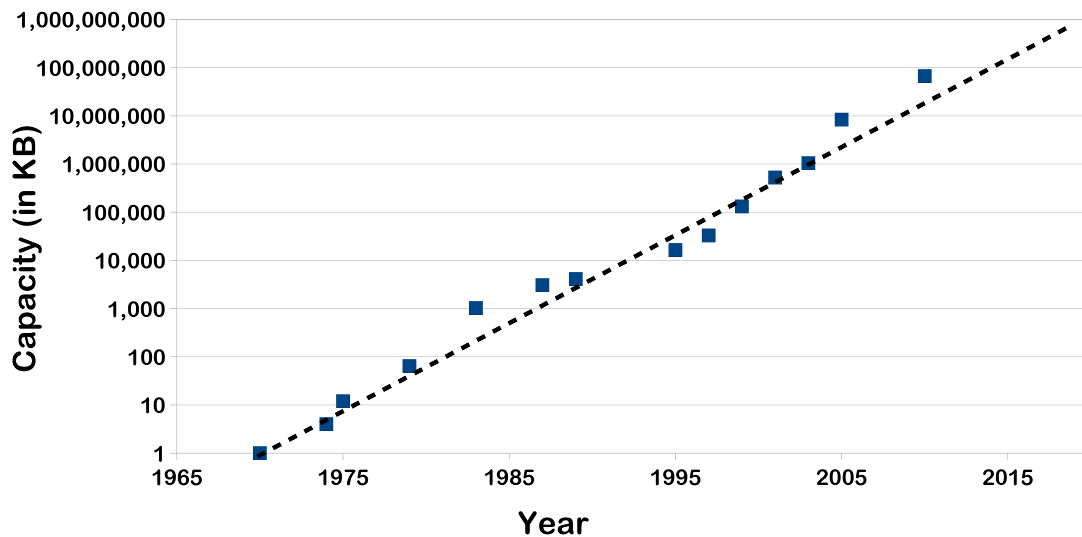


Figure 2.2: Historic trend of main memory (RAM) capacity. Survey of high-end desktops and low-end servers. *Sources* itrs.net, wikipedia.org, and www.jcmit.com/memoryprice.htm

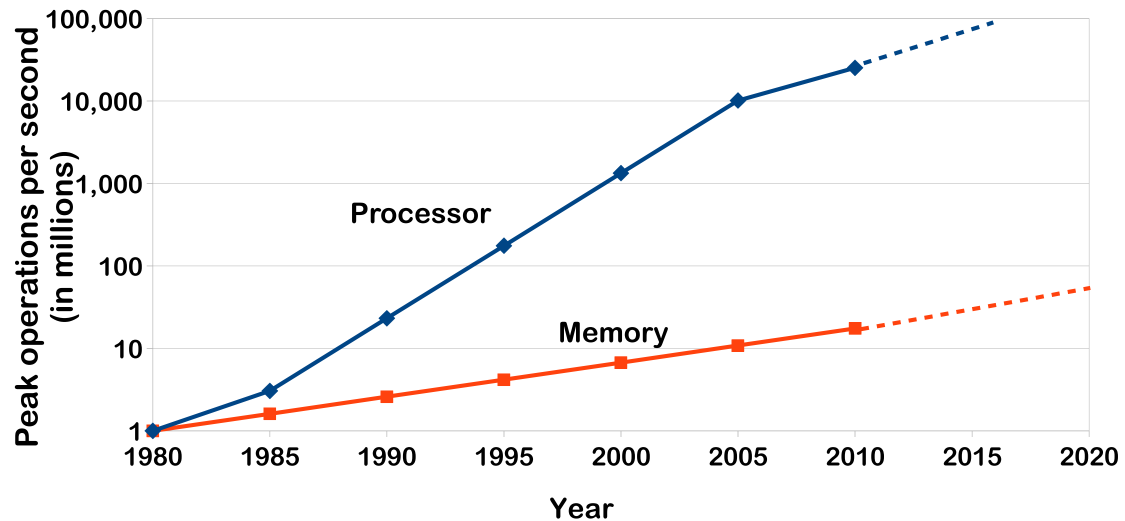


Figure 2.3: Performance gap between processor and main memory (DRAM). *Sources:* itrs.net, wikipedia.org, and Hennessy et al. [86].

computer components, processors and main memory. The trends displayed in the graphs, widely known as *Moore's law*, show exponential growth for both components [133, 134]. Recently, processors have surpassed one billion transistors on a single chip, while main memory sizes in the hundreds of gigabytes are becoming popular in server class computers. According to reports from the semiconductor industry, currents transistor scaling trends are predicted to continue until at least 2025 [57, 91].

Although transistor scaling allowed dramatic advances of both processors and main memory (DRAM), processors and memory have advanced in different ways. For the processor, the most significant implications of transistor scaling have been (1) increases in clock frequencies, and (2) the ability to design sophisticated architectures allowing for out-of-order and speculative execution. As transistor feature size shrinks, so does the switching time of each transistor, leading to opportunities to build processors with high clock frequencies (depicted in Figure 2.3).

In the case of DRAM, partly due to being a commodity component, the industrial focus has been on cost per bit [196, 200]. As a consequence, while the capacity of DRAM has followed the growth described by Moore's law, the access speed has not. The fact that off-chip accesses are both latency and bandwidth limited has resulted in a growing performance gap between processors and DRAM, as is shown in Figure 2.3. With current technologies, it is common to observe memory latencies of between 200 to 400 processor clock cycles.

Despite extensive academic and industrial research, the growing performance gap between processors and DRAM negatively affects the performance of a wide range of applications, as the introduced processor techniques have not been able to overcome the memory gap. In the following sections, we discuss some of the developments that have taken place to mitigate the performance impact of the memory gap.

2.1.2 Multicore Processors

Transistor scaling has enabled processors to be clocked at increasingly higher frequencies. From the early 1970s, until the mid 2000s, we have observed roughly 30% yearly increases in clock frequency. For example, the Intel 4004, released in 1971, was composed of 2300 transistors and was clocked at 740 kHz, while the Intel Pentium 4, introduced in 2000, incorporated 42 million transistors and a peak frequency of 2 GHz.

In the mid 2000s, however, the processor developments deviated from the three decades old trend. The ability to shrink transistor feature sizes no longer translated to as dramatic increases in transistor switching speeds. As can be seen in Figure 2.3, the current and future expected clock frequency improvements has decreased — from 30% a year before 2005, to less than 8% per year. The rate of switching speed increases was primarily influenced by limitations in the CMOS technology used, including the inability to further reduce voltage supplies, delays in interconnections, increase in power consumption and/or heat production, among others [81, 88].

A second major shift that occurred is the adoption of multiple processors integrated onto a single die, commonly known as *multicore* architecture. With the progress of transistor scaling, integrating billions of transistors into a single processor, and specifically allowing those transistors to yield improvements in software performance, has become challenging and costly. With multicore architectures, on the other hand, a doubling in transistor count can double the number of cores that can be built on a chip. As a result, processor manufacturers can offer the potential for doubling (parallel) software performance, with modest investments in chip design.

These two major shifts in mainstream processors, slower improvements in clock frequencies and the ubiquity of multicore architectures, has changed how computers are able to improve the performance of applications. Specifically, performance improvements that occurred due to processor microarchitecture[1] changes as well as clock frequency increases were mostly transparent to software. It was previously possible to execute the same software, with newer hardware, and observe a doubling in application performance. The current processor landscape changes this separation of concerns and requires that software be adequately parallelized in order to take advantage of newer processors.

Multicores were initially offered solely with homogeneous (same ISA, same microarchitecture) processing cores. However, in 2010 and 2011, various vendors have announced or introduced heterogeneous processing cores, typically organized as several general purpose cores, along with a single accelerator engine. The most popular architecture incorporates a graphics processing unit (GPU) as the acceleration engine; examples include the Intel's HD Graphics available in most of their chips today, AMD's Fusion, and Nvidia's Project Denver.

[1] In this text, processor microarchitecture refers to aspects of the processor implementation that are below the instruction set architecture (ISA) level and, therefore, transparent to the functionality of software.

Processor	Year	Cache levels	L1		L2		L3	
			size	latency	size	latency	size	latency
Intel 486 DX	1989	1	8 KB	1 cyc.				
Intel Pentium	1996	1	8–32 KB	1 cyc.				
AMD K5	1996	1	24 KB	1 cyc.				
AMD Athlon (K7)	1999	2	128 KB	1–3 cyc.	256 KB	18 cyc.		
Intel Pentium 4	2000	2	16 KB	1–2 cyc.	512 KB	20–25 cyc.		
AMD Opteron (K8)	2003	2	128 KB	1–3 cyc.	1 MB	8 cyc.		
Intel Core 2	2007	2	64 KB	1–3 cyc.	2–6 MB	10–14 cyc.		
Intel Nehalem *	2008	3	64 KB	1–4 cyc.	256 KB	10–12 cyc.	8–24 MB	50–60 cyc.
AMD Phenom II *	2009	3	128 KB	1–3 cyc.	512 KB	15 cyc.	6 MB	55 cyc.
Intel Sandybridge *	2009	3	64 KB	1–4 cyc.	256 KB	8–10 cyc.	3–15 MB	26–36 cyc.

Table 2.1: Cache hierarchy characteristics of x86 based processors in the past 20 years. Capacities of L1 caches represent the sum of the instruction and data caches, while L2 and L3 are unified caches, when present. Processors marked with (*) are multicore and the L1 and L2 sizes listed are *per core*. When an L3 cache is present, it is a single cache, shared by all cores.

2.1.3 Processor Caches, Buffers and Tables

Processor caches are the first line of defense against the memory performance gap. They are fast storage devices, significantly smaller than main memory, that are use for storing portions of main memory. The goal of processor caches is to reduce the average latency to memory by offering faster access to local copies of data. In general, the more accesses satisfied from the cache, the lower the average latency to access memory.

The first documented implementation of an on-chip cache (that is, a cache that is physically integrated within the processor die) is from the IBM System/360, model 85, in 1968 [122]. However, it wasn't until the second half of the 1980s that on-chip processor caches became a popular addition to mainstream processors [54, 181], because (1) as discussed in the previous section, the memory gap started to affect mainstream computers in the 1980s, and (2) the number of transistors available on single chips were reaching the 1 million mark, allowing for the integration of modest (8 to 16 KB) on-chip caches.

There are three main reasons why processor caches are significantly faster to access than main memory devices. First, they are built using the same transistor technology as processors, optimized for low switch time. Second, they are located on-chip, which allows accesses to avoid slow and bandwidth limited off-chip communication. Finally, they are significantly smaller, and can be structured to require fewer logic gate traversals and shorter wire transfers.

The processor industry has continued to incorporate increasingly larger caches in their processors, as both motivating trends for caches (the memory gap and abundance of transistors) have continued. In addition, because larger caches typically entail longer latencies, multiple levels of caches have been adopted in modern processors. Multi-level cache hierarchies include smaller, but faster, caches that are close to the processor, meant to satisfy a large portion of memory accesses, as well as larger caches, meant to satisfy a larger span of less frequently accessed items.

Table 2.1 lists several x86 based processors produced by Intel and AMD, detailing characteristics

of the on-chip cache hierarchy. While the list is not comprehensive, and other high-end architectures have incorporated larger on-chip caches, the information illustrates the availability of these caches in mid-range, popular processors. The Intel 486 DX, initially released in 1989, was the first x86 based processor to incorporate an on-chip data cache. Since then, a new cache level has been introduced to the on-chip cache hierarchy each decade. It is interesting to observe that, given the tradeoff between size and latency, each level of cache does not grow significantly throughout time.

Along with storage for caching memory, processors have adopted the use of on-chip storage for specialized uses other than caching main memory. These storage components, commonly referred to as tables or buffers, have been used to host branch prediction information, pre-decoded instructions, virtual memory translation information, and prefetching information, among others. These specialized storage devices are, in many cases, crucial for achieving good performance. For example, a recent study of high-performance computing (HPC) workloads on a 2008 AMD Opteron processor shows that inefficient use of translation look-aside buffers[2] (TLBs) can degrade application performance by up to 50% [129].

2.1.4 Prefetching and Replacement Algorithms

A principal metric for determining the utility of on-chip caches is the improvement, in terms of efficiency, of application execution. Given the potential impact of caches in application performance, as described in the previous section, significant research has been conducted with the goal of improving the utility of the different types of caches found on modern processors. Two principal avenues for improving cache utility have been prefetching algorithms and replacement policies.

Prefetching algorithms are used to retrieve items currently not in the cache, before they are requested. Prefetching algorithms try to predict which memory items currently not in the cache are *most* likely to be accessed in the near future. If accurate in their predictions, both replacement and prefetching techniques can increase the utility of caches by reducing the number of times items are not found (missed) in the cache. The replacement policy, on the other hand, determines which of the current items in the cache should be evicted to make space for a new item. In essence, replacement policies try to predict which of the currently cached items are *least* useful (least likely to be accessed in the near future).

One source of inefficiency that affects the performance of modern processor caches is **cache pollution**. Cache pollution can be defined as the displacement of a cache element by a less useful one. In the context of processor caches, cache pollution occurs whenever a non-reusable cache line is installed into a cache set, displacing a reusable cache line, where reusability is determined by the number of times a cache line is accessed after it is initially installed into the cache and before

[2] Translation look-aside buffers (TLBs) are used as a fast cache of most recently accessed page-table entries, which are to used to perform per process translation of virtual addresses to physical addresses. Typically, hardware or software traversals of page-tables is a long latency operation, potentially requiring several memory accesses. The fast access of translation information through TLBs have allowed caches to be physically and/or indexed, without significantly affecting access latencies.

its eviction.

In the remainder of this section, we will summarize the development of prefetch and replacement algorithms in processors. In particular, we highlight the problem of cache pollution in each case and review previous research proposals targeted at reducing the impact of cache pollution.

2.1.5 Prefetching

Hardware prefetchers, for both instructions and data, have been incorporated into all mainstream processor designs. These prefetchers monitor the access patterns of programs and use this information to request chunks of memory that may be used in the near future. When memory accesses are easy to predict, prefetching can be effective in reducing the impact of the memory gap in application performance. For example, Can and Nagpurkar analyzed the prefetchers used in the IBM's POWER6 processor, released in 2006, and found that certain workloads observed performance improvements of up to 350% [38].

Early work in prefetching was summarized by Alan Smith in 1978 [179, 180]. The consensus, with the technology at the time, was that the only prefetch strategy feasible was the *one block lookahead* (also known as next-line prefetcher). Since then, given the abundance of transistors in a die, along with the increase in the memory performance gap, processors have adopted more complex prefetch strategies [14, 38, 79, 100, 145, 152, 176, 182, 190]. Along with data and instruction prefetching, researchers have also explored prefetching in the context of other processor structures, such as TLBs, demonstrating performance improvements for workloads that exercise the processor's memory management unit (MMU) [53, 92, 102, 167].

One fundamental challenge in designing hardware prefetchers, which relates to the inaccurate nature of prefetch predictions, is that of tuning *prefetch aggressiveness*. On the one hand, increasing the number of prefetch requests or making requests earlier in time (i.e., making the prefetcher aggressive) will likely increase the number of memory requests that are serviced through prefetching. On the other hand, increasing the aggressiveness also increases the number of useless prefetches (i.e., items that are not used by the time they are replaced). Useless prefetching increases the amount of cache pollution observed, and negatively impacts cache performance, potentially negating the performance gains of useful prefetches [184].

To overcome the possible pollution effects of prefetching, Jouppi proposed to separate the cache storage from the storage used for prefetching items, typically called *prefetch buffer* [100]. Mutlu et al. proposes that the L1 cache act as the prefetch buffer; where prefetched cache lines are initially inserted only in the L1 cache and inserted into the other levels of the cache hierarchy only if they are accessed before being replaced. This strategy does not prevent pollution at the L1 level, but ensures that lines at the other levels have been accessed at least once [138]. Another venue to reduce pollution due to aggressive prefetching is to introduce an independent prefetcher filter that controls which requests made by the prefetcher are actually sent to the cache hierarchy [121, 210]. Instead of modifying the logic of the prefetcher itself, the filter keeps track of the usefulness of the

different types of prefetch requests made by the prefetcher. The filter subsequently may decide to eliminate specific prefetch requests that are predicted to be useless and pollute the cache.

2.1.6 Replacement

The simplest replacement policy used in hardware structures is based on a direct-mapped structure. In this scheme, a mapping function is used to associate the identifier key (typically an address) to a singular location in the cache. Its simplicity comes from the lack of per item metadata, and the low number of operations to determine membership and to determine the item to be displaced. In practice, however, direct-mapped caches observe a large number of *mapping conflict misses*, due to various distinct items that are used concomitantly being mapped to the same cache location [40, 87].

To overcome the performance degradation caused by mapping conflict misses, computer architects have introduced cache organizations that allow identical mappings to occupy multiple locations in the cache (known as *sets*). The number of positions available in each set is known as *associativity*. Although this design requires metadata for deciding what to evict, and more logic to determine membership than the direct-mapped case, it has been increasingly adopted in cache designs because of its superior hit-rate performance.

Despite the extensive body of work on replacement policies in the context of both processor caches and virtual memory, the least recently used (LRU) algorithms, and derivatives, are the most widely adopted replacement policies. There are two main reasons behind LRU's wide usage. First, LRU has proved to be effective at achieving high hit-rates for a wide range of applications and access patterns. Second, there have been various proposals that approximate LRU with a simple and efficient implementation (e.g., CLOCK [58] for virtual memory and Pseudo-LRU [180] for caches).

In the past decade, there has been renewed interest in research of replacement policies for on-chip processor caches, not only because of the growing memory performance gap, but also because of the complexity of access patterns of modern applications that do not conform well to LRU based caching [19, 128]. An added incentive for research in processor cache replacement policies is the growth in the number of levels in the cache hierarchies. As shown in Table 2.1, while in the 1990s a single level of modest sized caches was present, today processors typically boast 3 levels of caches with capacities reaching that of the entirety of main memory of 1990s computers.

Numerous studies have proposed enhancing the LRU cache replacement policy to avoid cache pollution [73, 85, 96, 105, 120, 123, 150, 166, 203, 205]. These studies attempt to augment LRU replacement decisions with information about locality, reuse and/or liveness. For example, the dynamic insertion policy, proposed by Qureshi et al., focuses on adapting the initial *placement* of caches lines in the LRU stack of each cache set, depending on the application access pattern [154]. The proposed dynamic insertion policy (DIP) reduces competition between caches lines by reducing the time to eviction of cache lines with thrashing access patterns.

2.1.7 Eliminating Mapping Conflict Misses in Direct-Mapped Structures

Numerous efforts have focused on avoiding *mapping conflict misses* in direct-mapped caches, both at the L1 and L2 levels [28, 37, 55, 104, 117, 125, 127, 137, 163, 164, 175]. In direct-mapped caches, mapping conflict misses have a relatively high probably of occurring; this happens whenever different cache-lines that map to the same position of the cache are reused and show temporal locality. Even if the cache has capacity to hold all concurrently accessed lines, since lines map to the same location, they may still cause misses. Most solutions, whether dynamic or static, attempt to predict or detect application memory access patterns to create mappings that minimize mapping conflict misses.

Bershad et al. explored dynamically avoiding mapping conflict misses in large direct-mapped caches [28] and they proposed a small hardware extension called *Cache Miss Lookaside buffer* (CML) which records a fixed number of recent caches misses. The operating system uses this buffer to identify pages that map to the same cache partition and are used concurrently. The identified pages are remapped to other partitions with low miss count, by means of page copying.

Romer et al. extended the work by Bershad et al. by eliminating the need for a CML [164]. In essence, a software-filled TLB is used to monitor accesses to cache conflicting pages. If the distance between TLB refills to cache conflicting pages is small, one page is chosen to be remapped. While performance improvements were shown in some workloads, Romer et al.'s TLB Snapshot-Miss policy showed greater overheads and less accuracy then the CML hardware.

In recent years, however, direct-mapped caches have not been used in many general-purpose, or high-end processors. Hardware manufactures have invested in producing high-associativity caches to resolve mapping conflict misses. Typical L1 associativity for modern processors range from 2 to 8, while L2 associativity range from 8 to 16. Even larger associativities are used for higher-level caches, or off-chip caches.

2.1.8 Cache Bypassing

Previous research most similar to the work presented in Chapter 3 is the study of *cache bypassing* to eliminate cache pollution. Cache bypassing consists of refraining from installing selected cache lines into the cache, or, at least, one of the levels of the cache. If cache lines that are not re-accessed in the near future are chosen for bypassing then this strategy improves overall cache utilization, since reusable lines are less likely to be prematurely displaced from the cache.

The majority of work exploring cache bypassing has focused on reducing cache pollution in the first level cache (L1) [49, 99, 191, 204]. There appears to have been little work that explores cache bypassing for L2 cache [68, 106, 149]. All the studied dynamic schemes require hardware support and propose non-trivial changes to the processor and cache architecture.

Dybdahl et al. claim to be the first to explore cache bypassing at the last level cache (the cache level closest to memory, which is typically the largest and slowest cache of the cache hierarchy) [68].

They extend previous work done by Tyson et al. for L1 cache bypassing [191], adapting it for use with a physically indexed L2 cache. Tyson et al. determined the empirical relationship between candidate cache lines for bypassing and specific load/store instructions. In this scheme, a table is dynamically built based on instructions that generate more cache misses than hits.

The application of Tyson's scheme to the last level cache resulted in both reduction and increases in the miss ratio, depending on the workload; miss ratios of SPEC CPU 2000 benchmarks varied from a 58% reduction up to a 132% increase. Dybdahl et al. proposed enhancing the Tyson scheme by augmenting every L2 cache line to include extra information for dynamically tracking the potential for cache bypassing [68]. The combined instruction table and L2 cache line miss information attenuated the performance degradation, and, unfortunately, the improvements as well. Miss ratios for the same set of benchmarks varied from a 50% reduction up to a 37% increase.

Piquet et al. propose classifying "single-usage" cache-lines for improved L2 cache replacement policy and bypassing [149]. In their work, they also investigate the relationship between specific instructions and caching behavior. They propose the creation of the *block usage predictor* table, stored in main memory, along with augmenting the L2 tag with *single-usage* and instruction address information. With this information, they enable cache bypassing for specific instructions. In addition, the LRU replacement policy is modified to first consider replacement of likely singe-usage lines. In their simulations, they observe instructions-per-cycle[3] (IPC) increases of up to 30% in memory intensive workloads from SPEC CPU2000, as a result of a 35% decrease in L2 miss rate.

More recently, Kharbutli and Solihin have proposed a similar cache replacement and bypassing scheme to that proposed by Piquet et al., but exploring different metrics to guide replacement and bypassing [106]. Kharbutli and Solihin propose enhancing L2 cache tags with "counter-based" information, and explore metrics such as reuse (access interval and live-time) to predict lines that should be replaced from the L2 cache, or not inserted at all. They observed performance improvements of up to 48% on a memory intensive application from the SPEC CPU2000 suite.

It is interesting to note that most existing processors have a rudimentary cache bypassing mechanism that implements *full* cache bypassing, primarily for the purpose of interacting with external devices. It is commonly referred to as cache inhibited and I/O mapped memory. The main difference between this existing mechanism and the proposals described above is that the proposals typically require partial bypassing (at either the L1 or L2 cache level); not full bypassing, where the cache hierarchy is bypassed altogether. This full bypassing mechanism, however, has not been successfully explored for minimizing replacement misses. As far as we know, there have been no studies reporting the use of cache inhibited memory for eliminating cache pollution. The most likely reasons include: (a) the cache-inhibited attribute is usually implemented as part of the memory management unit and, hence, is applied at page granularity; and, (b) most previous work has

[3]Instructions per cycle (IPC) is a widely adopted metric for measuring efficiency of processor execution. Given the same stream of instructions, the higher the number of instructions that are executed every cycle, the faster the overall execution is.

dealt with cache pollution at either L1 or L2 level caches, but not of the entire cache hierarchy.

2.2 Computer System Software

Computer system software, or simply *system software*, is the software responsible for operating the computer hardware as well as to provide a platform for higher-level software, such as applications and middleware. Typical examples of system software programs include the BIOS, operating system, compiler and run-time libraries. The evolution of system software, particularly operating systems, has been largely reactionary, changing in response to advances in hardware technology and application.

In this section, we introduce key concepts of operating systems that relate to our work. We also discuss recent operating systems proposals for enabling efficient execution of applications, as well as performance optimizations in the context of run-time and operating systems

2.2.1 Virtualization and OS Abstractions

Perhaps one of the most fundamental mechanisms used in operating systems to expose (hardware) resources to applications is *virtualization*. Virtualization is the process of offering a resource to software without directly offering access to the *physical* resource — instead, a *virtual* version of the resource is offered. Examples of resources that have been successfully virtualized include (1) processors, which has led to abstractions such as processes and threads, (2) memory, allowing applications to use multiple hardware resources (e.g., memory and disk) transparently, through a uniform interface, (3) storage, which has led to the concept of file-systems that are significantly more feature rich than raw disks, and (4) network devices, hidden behind a simpler *socket* interface.

Virtualization of hardware was initially adopted as a mechanism to allow multiple applications, or users, to efficiently share the same physical machine (until the era of the personal computer, machines were costly and a scarce resource). With the evolution of computers, virtualization was also found to be a valuable mechanism to allow the independence between hardware and software. As long as the virtual resource was largely independent from the its physical counterpart, the same abstractions could be used by operating systems to support different, or newer, hardware. One example are the file-system abstractions that have been successfully used to provide storage services for several decades, and have been largely unchanged despite the diversity of storage devices (e.g., hard-disks, floppy disks, CD-ROMs, network storage, flash drives).

The abstractions and services offered by operating systems, as well as their implementation, play an inordinate role in the functionality, reliability and performance of a computer system. As such, operating systems researchers and designers have experimented with how best to implement various services. For example, the exploration of different design principles for offering operating system services have led to the proposal of different kernel architectures such as monolithic, microkernel and exokernel [1, 70, 71, 115, 116]. These architectures differ on *where* services should be

implemented (whether they should be offered by the core kernel, or offered separately), and *how* the services should be implemented (in a distributed way in each of the application's address space as a library, or as an external server process, as a centralized service).

2.2.2 Support for Parallelism

Parallelism has been extensively used to improve performance of computer systems. At the processor level, parallelism is used in techniques such as instruction pipelining, vector and superscalar execution. At the software level, multiple processing engines can be used to enable task-level parallelism. At larger scales, involving multiple computers, parallelism is also used to enable supercomputers and data-center computing. In this section, we will introduce concepts and previous work in operating system support for task-level parallelism.

Operating system work in supporting parallelism can be roughly classified into two categories: (1) structuring the internals of the operating system kernel to be scalable, (2) providing services to applications to scale with increased concurrency. A thorough summary of operating system research to support multiprocessor execution can be found in Chapter 2 of the dissertation by Jonathan Appavooo, published in 2005 [8].

Tornado and subsequently K42 were among the first operating systems to argue that there are performance advantages in designing operating systems specifically for multiprocessors and to maximize locality and independence of execution [9, 10, 83]. After 2005, research in systems software support for parallelism has regained attention due to the ubiquity of multicore processors.

The Corey operating system advocates requiring applications to explicitly specify which resources should be accessible to more than one processor [33]. The principle behind this requirement is that many resources (e.g., portions of memory, file descriptors, etc.) are used by only one processor, and can be optimized by being implemented without concurrency support. For the resources that are declared to be shared among multiple processors, a parallel implementation is used instead.

The Factored Operating System (fos) uses a micro-kernel architecture, with services implemented as different processes, to schedule specific system services onto dedicated cores [199]. The goal of dedicating system services to cores is to improve the performance of these services by avoiding competition of on-chip storage (caches, TLB, etc.) between application execution and the execution of operating system services.

The Barrelfish operating system, which exemplifies an operating system that uses a *multikernel* structure, investigates a distributed, message-passing, structure to operating systems, even when executing on a shared-memory computer [20, 21]. Within the operating system, no state is implicitly shared among multiple cores and messages must be exchanged to update state among cores. The underlying argument for promoting a distributed, multikernel design for an operating system is that current and upcoming computers have begun to resemble a distributed system specifically with respect to performance characteristics. Consequently, having the system software match the

underlying architecture allows for more efficient computation and communication.

Helios is a proposal to simplify the development and the running of applications on computers with heterogeneous processors [143]. It proposes partitioning the operating system into a main kernel, and a series of *satellite kernels* that only communicate through message-passing, similar to the multikernel architecture. The goal of Helios is to allow applications to rely on the same operating system services that are available on the general purpose processors, even when executing on accelerators such as graphics processors (GPU) and programmable network processors.

The Tessellation operating system advocates partitioning both the operating system and application into *Cells* to promote locality [56, 124]. Each component belonging to a Cell is scheduled concurrently, similar to gang scheduling, to achieve what the authors call *Space-Time Partitioning*. Similar to the multikernel design, inter-Cell communication occurs through message-passing primitives. Also, Tessellation uses a two-level scheduler to allow each Cell to control the use of its resources independently from the operating system.

Recently, the MIT group behind the Corey operating system revisited the assumption that novel operating system architecture are necessary to efficiently support multicore processors [34]. By analyzing the performance of a series of workloads running on Linux, a monolithic kernel that has slowly evolved to support multiprocessing over the past 15 years, and applying modest changes to the kernel, they conclude that a traditional operating system architecture is able to scale to a large number of cores in a multicore computer system.

2.2.3 I/O Concurrency: Threads and Events

A different type of parallelism than multiprocessing that is also crucial to performance of modern computers is I/O concurrency. Because the latency of I/O devices are orders of magnitude slower than processor speeds, but can exhibit high throughput rates (enough to saturate several processors), a large number of requests should be submitted to the device in order to guarantee full bandwidth utilization of the device. To this end, software must produce a large number of *concurrent* I/O requests.

Applications that are required to efficiently handle multiple concurrent requests rely on operating system primitives that provide I/O concurrency. These primitives typically influence the programming model used to implement such applications. The two most commonly used models for I/O concurrency are: (1) blocking threads and (2) non-blocking/asynchronous I/O with subsequent notification (this latter model is commonly referred to as event-driven architecture).

Thread based programming (with blocking) is often considered the simplest, as it does not require tracking the progress of I/O operations (which is done implicitly by the operating system kernel) [194]. A disadvantage of threaded servers that utilize a separate thread per request or transaction is the inefficiency of operating with a large number of concurrent threads. The two main sources of inefficiency are the extra memory usage allocated to thread stacks and the overhead of tracking and scheduling a large number of execution contexts [198].

To avoid the overheads of threading, some developers have adopted the of use event-driven programming. In an event-driven architecture, the program is structured as a state machine that is driven by progress of certain operations, typically involving I/O. Event-driven programs make use of non-blocking or asynchronous primitives, along with an event notification systems to deal with concurrent I/O operations. These primitives allow for uninterrupted execution that enables a single execution context (e.g., thread) to fully utilize the processor. The main disadvantage of using non-blocking or asynchronous I/O is that it entails a more complex programming model. The application is responsible for tracking the status of I/O operations and availability of I/O resources. In addition, the application must support multiplexing the execution of stages of multiple concurrent requests.

In both models of I/O concurrency, the operating system kernel plays a central role in supporting applications in multiplexing the execution of concurrent requests. Consequently, to achieve efficient server execution, it is critical for the operating system to expose and support efficient I/O multiplexing primitives. For an extensive performance comparison of thread-based and event-driven Web server architectures, under modern UNIX I/O primitives, we refer the reader to work by Pariag et al. [146].

In Chapter 5 and 6 of this dissertation, we explore novel operating system support for thread-based and event-based programs. In the remainder of this section, we discuss some of the influential and related work that studied operating system support for these two models of I/O concurrency.

Perhaps the most influential work in thread based support for I/O concurrency is Scheduler Activations [7]. Anderson et al. identified problems with kernel-mode and user-mode threading models. The problem highlighted for kernel-mode threads is that they involve a fundamental overhead of switching in and out of kernel mode in order to performance a switch — an overhead that is particularly prevalent in I/O intensive applications. User-mode threading, on the other hand, allows for fast switches and are easy to customize. However, because these threads are transparent to the operating system, scheduling events due to I/O, multiprogramming and page faults can inadvertently introduce idle time. To overcome this issue, Scheduler Activations proposes returning control of execution to a user-mode scheduler, through a scheduler activation (akin to a new kernel visible thread) upon experiencing a blocking event in the kernel. This way, threads can be switched cheaply in user-mode, while not suffering unnecessary idle time.

The work described in Chapter 5, in which we present a threading package that uses user-level threads in conjunction with kernel visible threads, was partially inspired by Scheduler Activations. However, our solution to addressing the problem of the operating system inadvertently blocking kernel visible threads that have user-level threads ready to execute is different. In our approach, operating system and application execution is *decoupled*, and execution of the both components occur independently.

Adya et al. proposed the use of cooperative user-level threads (*fibers*), in conjunction with auto-

matic stack management, to construct highly concurrent applications [3]. Their goal was to enable the ease of reasoning of thread based programming, while allowing I/O operations to be multiplexed through event-based primitives. Consequently, in their system, both thread-based and event-driven abstractions are exposed to the programmer. Threads are implemented as user-level fibers and managed through a run-time library. When a thread issues an I/O operation, the operation is transformed, by the library, into an asynchronous I/O request with an attendant continuation that will activated once the I/O operation completes.

In the work described in Chapter 5 (FlexSC-Threads), we also propose the use of cooperative user-level threading to extract independent system call requests from threaded applications. While the run-time libraries for both the *fibers* threading and our proposal should be quite similar, the main difference between these two systems are with respect to interfacing with the operating system. The work by Adya et al. relies on the use of asynchronous I/O operations, while in our proposal we use exception-less system call interface. As we show in Section 6.4 the use of exception-less system calls yields improved performance when compared to asynchronous I/O operatiions. Also, in Section 6.1 we discuss other advantages of exception-less system calls over asynchronous or non-blocking I/O operations.

Behren et al. presented another well known system, called Capriccio, that promotes thread based architectures for I/O intensive applications [195]. In Capriccio, several techniques were introduced to reduce the overheads of threading in the context of high I/O concurrency. Specifically, Behren et al. argued that there are inherent performance benefits to user-level threading, and also described how to efficiently manage thread stacks, minimizing wasted space, and proposed resource aware scheduling to improve server performance. Through these techniques, Behren et al. experimentally showed that internet servers built with Capriccio could achieve comparable performance to equivalent servers that used an event-driven architecture.

The techniques described by Behren et al., while also targeting performance improvement of server type applications, are largely orthogonal to the work presented in Chapter 5. While Capriccio introduced novel techniques to improve user-level threading, it did not address performance problems relating to the direct interaction of application with the operating system. In effect, we believe it would be possible to integrate most of the techniques proposed in Capriccio to the threading library we describe in Chapter 5.

The Flash web server project [144] combines the use of both thread based programming and event-driven primitives. In the proposed architecture, event driven primitives are used for requests that only require data that is cached in memory; for requests that may block, separate helper threads are used.

Banga et al. are among the first to identify the lack of scalability in traditional UNIX event delivery mechanisms (*select* and *poll*) [16]. Subsequently, they explored the construction of generic event notification infrastructure under UNIX, with the goal of scaling to a large number of monitored (network) file-descriptors, as network servers were becoming increasingly popular at the

time [17]. Instead of relying on a state-less interface, such as *select* and *poll*, the new event delivery mechanism was stateful and required applications to notify the kernel of *changes* in the events to be monitored (and not the entire list of events to be monitored). It also featured a scalable queue based interface to process incoming event notifications.

The work by Banga et al. inspired the implementation of the *kqueue* interface available on BSD and Linux kernels [113]. The *kqueue* interface not only tried to improve the scalability of event based notifications, but also its generality. Interfaces such as *select* and *poll* can only operate on file-descriptors, and consequently on resources that are virtualized by file descriptors. With the *kqueue* interface, it is also possible to monitor several other types of events such as signals, memory, etc.

Elmeleegy et al. proposed lazy asynchronous I/O (LAIO), a user-level library that relies on Scheduler Activations to support event-driven programming [69]. With LAIO, since it is based on scheduler activations, any system call is supported and allows the operating system to provide the traditional system call interface. In their work, they argue that lazily spawning threads when operations block in the kernel, and only notifying the application when the operation completes, and not in an intermediate stage, makes it easier for programmers to use asynchronous I/O.

Recently, the Linux community has proposed a generic mechanism for implementing non-blocking system calls, similar to LAIO [36]. In their proposal, if a blocking condition is detected, they utilize a "syslet" thread to block, allowing the user thread to continue execution.

In Chapter 6, we propose a new way of supporting event-driven applications, that addresses some of the shortcomings of previous work . We believe our proposal has all the advantages of the studies to better support event-driven programs described in this section ([16, 36, 69, 113]), in terms being general purpose (i.e., supporting access to any type of operating system managed resource) and minimally intrusive operating system implementation. However, our proposal uniquely provides performance advantages due to the ability to reduce processor state pollution that results from multiplexing application and operating system execution.

The staged-event driven architecture (SEDA) was proposed for designing highly concurrent Internet services, highlighting the robustness of the architecture in overload conditions [198]. The SEDA approach divided server code into separate stages, which are then connected through event queues. Through resource throttling of different stages, SEDA is able to achieve high performance and graceful degradation of performance under high loads.

Krohn et al. try to address the programmability of event-driven servers in Tame [110]. Tame provides language-level constructs that hide details of requesting I/O and subsequently waiting for event notification, as well as communication between stages of the program.

One added challenge of event-driven programming, specially when compared to thread based programming, is efficient use of multiprocessors. One commonly used technique is to implement a worker thread poll pattern, where a single master thread enqueues requests to worker threads. While this allows for multiple independent threads to be scheduled on multiple processors, it only works well for applications that do not have significant amounts of synchronization or data sharing.

To address the multiprocessor challenge, Zeldovich et al. introduced *libasync-smp*, a library that allows event-driven servers to execute on multiprocessors by having programmers specify events that can be safely handled concurrently [207]. *Libasync-smp* requires programmers to specify program stages that do not have data dependencies, allowing the library to concurrently schedule execution of independent stages, while requiring no new operating system support.

2.2.4 Locality of Execution and Software Optimizations for Processor Caches

The "principle of locality" has been the guiding principle behind caches, both within virtual memory and processor caches [60, 61, 62]. The principle stems from the observation that, for several classes of programs, once a set of instructions or data are accessed it is likely that the items in that set will soon be accessed again for a period (phase) of execution. While some programs naturally exhibit highly localized accesses, and consequently make efficient use of caches, other programs exhibit lower degrees of locality.

Due to the lack of inherent locality in some programs, there has been a large body of work in transforming execution to produce more localized accesses. These transformations have been explored at many different levels of the software stack, including in the context of theory of algorithms [78, 172], language-level constructs [112, 165], static compiler optimizations [39, 41, 50, 51, 103, 111, 201], as well as run-time and operating system optimizations [2, 30, 74, 169, 187]. In the remainder of this section we highlight research that is specific to run-time and operating systems.

Bellosa et al. propose using feedback, either through hardware performance counters or TLB misses, to determine threads that potentially share data [22, 23]. This information is used to determine how threads should be ordered in the run-queue so as to minimize cache misses that occur immediately after context switches. Weissman also experimented with identifying thread locality at run-time using hardware performance counters [197]. One main difference in the work by Weissman is the requirement for programmers to provide annotations about the state sharing between threads.

Larus and Parkes were among the first to recognize that although modern server workloads exhibited poor locality of accesses, it was not inherent to the server program, but partly due to how it was scheduled [112]. They used this insight to proposed Cohort Scheduling to improve locality of server workloads. Cohort Scheduling requires programmers to divide the program into stages, using a *StagedServer* library. Stages are then scheduled with the goal of maximizing data and instruction locality, with hints from the application developer on how stages are expected to interact or interfere with each other.

Tam et al. proposed *thread clustering* to improve the cache reuse in the presence of multiple application threads sharing cache content [187]. The thread clustering technique uses hardware performance counters, at run-time, to identify ranges of the virtual address space that are accessed by each thread. This information is then used to infer which threads have data access affinity, and should consequently be scheduled concurrently in a chip multiprocessor (CMP) system to

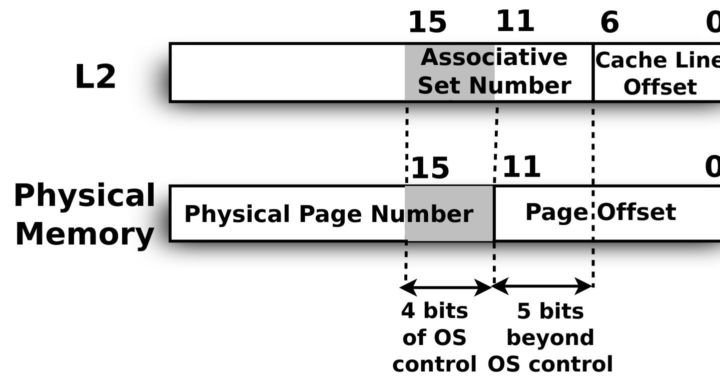


Figure 2.4: Cache indexing of physical addresses in the PowerPC 970FX processor, with 128B cache lines, L2 cache with 512 sets, and 4KB pages.

maximize sharing.

Software cache partitioning has been used in different ways to reduce the interference of executing multiple processes; these techniques are discussed in more detail in Section 2.2.5. Also, a specific type of optimization to improve cache utilization is to reduce the interference between application and kernel execution; recent proposals to address this type of interference are discussed in Section 2.2.6.

2.2.5 Page Coloring and Software Cache Partitioning

Modern caches employ a simple hashing function for the purpose of cache line indexing. Lower-level caches generally use the virtual address provided by memory requests, as these accesses are time-sensitive and cannot afford to be translated to physical addresses in the critical path. Higher-level caches, which are larger and slower, are generally physically indexed. In physically indexed caches, physical addresses of data are used to map data into cache sets. The hashing function used for indexing into the cache must utilize enough bits from the address to be able to index anywhere into cache. Due to the relatively large size of current caches, the number of bits required is larger then the number of page offset bits, as shown in Figure 2.4. As a consequence, the choice of virtual to physical mapping influences the specific cache sets which store application data.

Conceptually, the set of pages which share the indexing bits above the page offset form a congruence class. Each congruence class is mapped to a fixed partition of the cache. The number of congruence classes available is equal to 2^n, where n is the number of bits used for cache hashing above the page offset. In the case of the processor depicted in Figure 2.4, the PowerPC 970FX, the L2 is organized into 512 sets, resulting in $\log_2(512) = 9$ bits used for indexing. The four most significant of those bits can be used to determine congruence classes. As a consequence, the operating system can control $2^4 = 16$ different cache congruence classes or *partitions*.

The influence of page selection on cache indexing has been observed and studied since the late 80's and early 90's [104, 125, 177]. These studies prompted the concept of *page coloring*. Page coloring consists of classifying pages that belong to the same cache congruence class and, there-

fore, map to the same portion of the cache. Studies have modified operating system page allocators to use page coloring, showing performance improvements due to reduced mapping conflict misses [37, 104, 117].

In the late 90's, the same technique was used for software cache partitioning. If the page allocator enforces the use of the same cache congruence class (page color), portions (or all) of the application virtual address space can be restricted or isolated to a subset of the processor cache. In essence, each cache congruence class is deterministically mapped to a subset, or partition, of the cache. In this view, the software system can manipulate each of the different possible cache partitions through careful page allocation.

The first application of software cache partitioning through page coloring was presented by Andrew Wolfe [202]. In his work, Wolfe proposed partitioning the cache to achieve predictable performance in preemptible real-time systems. By ensuring that the cache content of preempted real-time applications remains intact, the partitioning minimized the interference of preemption on the application's performance. Software cache partitioning for real-time systems has gained attention by the community in subsequent work [118, 136, 159, 192].

In recent years, there has been a resurgence of software cache partitioning due to the commercial impact of chip-multiprocessors (CMP). Many chip-multiprocessor configurations produced are designed with *shared* higher level caches (L2/L3). Due to sharing, applications can interfere with each other, potentially affecting the performance of the applications involved [43, 186, 188]. The computer architecture community has been actively studying and proposing alternatives to attack the observed interference; however, a no solution has been implemented of commercial hardware. Software cache partitioning has been proposed as a solution to shared cache interference [52, 156, 185, 186, 188]. Although software based cache partitioning is less flexible and incurs higher overhead than hardware solutions, it is very attractive as it can be implemented on current, widely available CMPs.

As we discuss in Chapter 3, we use page-coloring and the ability for the operating system to divide the cache into partitions as the basis of the *software pollute buffer*. A significant difference with the previous work described in this section, and the work described in Chapter 3 is that previous have used cache partitioning to isolate performance interference between applications (i.e., inter-application interference). In our work, we show how cache partitioning can be used to improve caching of a single application, by reducing *intra-application* interference.

2.2.6 Operating System Interference

In Chapter 4, we revisit the topic of the effects of operating system execution on application performance, but from the perspective of the processor and attendant memory and cache hierarchies. We have gained much insight from previous studies on this topic [4, 11, 45, 42, 130]. Specifically, server applications, which make extensive use of operating system services such as I/O, memory allocation, and inter-process communication, are known to be more sensitive to the performance of

operating systems kernels. In Chapter 4, we study the impact that the multiplexing of application and operating system execution has on overall performance. We focus on analyzing the impact of this type of execution on the processor, with detailed information about the effects of various performance sensitive processor structures. We believe the analysis we provide brings new insights to the issue of operating system interference on application performance.

With the popularity of high performance network devices in the 90s, the implementation of networking within the operating systems started to become a bottleneck. In particular, interrupt based processing and scheduling was shown to lead to performance degradation or even livelock the operating system [67, 132]. Druschel et al. propose the use of lazy receiver processing (LRP) to minimize both the amount of interrupt handling and context switches between the operating system and application [67]. Mogul et al. propose the use of both interrupts and polling to eliminate potential livelock [132]. By default, interrupts are used to process incoming requests, however, if a certain interrupt rate threshold is reached, their system disables interrupts and falls back to polling.

Mohit and Druschel proposed a new operating system facility, called Soft Timers, that enables cheap polling at microsecond granularity [11]. This facility proved to improve performance of network processing. Other researchers have also explored mechanisms to reduce interference of network processing on application execution [5, 35, 76, 158, 173]. The basic mechanisms behind the Soft Timers facility, namely, removing interrupts in the favor of periodic polling, along with batching device driver execution, have inspired the work presented in Chapter 4 (exception-less system calls). While the Soft Timers technique addressed the interference of I/O interrupts, along with the attendant execution of the interrupt handler, our work deals with the other principal source of entry to the operating system: the system call interface. In practice, we believe operating systems should support both techniques to support asynchronous execution of operating system work, whether triggered by I/O devices or application requests.

Computation Spreading, proposed by Chakraborty et al., utilizes the abundance of on-chip resources of modern multicores by allowing the hardware to automatically migrate computation, to specialize cores [42]. They introduced processor modifications to detect similarity of execution of different threads and to allow for hardware migration of threads. The evaluation of their work focused on separating user and kernel execution onto different cores. Their findings indicate that application and operating system execution cause negative interference in various performance critical processor structures.

Explicit off-loading of select OS functionality to dedicated cores has also been studied for performance and power reduction in the presence of single-ISA heterogeneous multicores [140, 141, 131]. One challenge of these proposals is that they rely on expensive inter-processor interrupts (IPI) to offload system calls, making it difficult for potential improvements in locality to outweigh the overhead of the inter-processor interrupt.

At a high-level, the exception-less system call proposal, particularly the thread-based version of our proposal (Chapter 5), is similar to Computation Spreading in that they both advocate for

scheduling application and operating system work on separate cores in order to improve locality of execution. However, we believe that scheduling, particularly multi-processor scheduling, and system call execution are fundamentally tasks that should be performed at the operating system level. There are a few techniques employed by Computation Spreading to overcome the fact that it is implemented at the hardware level and these techniques limit the applicability of Computation Spreading. The requirement for these techniques indicate that the operating system layer is better suited to implement scheduling of operating system work. For example, Computation Spreading requires that processors implement several hardware threads in order to migrate virtual CPUs from one physical core to another transparently to the operating system, without leaving some cores idle. Further, migrating threads on *every* mode switch is impractical to do on modern hardware. The current mechanism for inter-core notification, inter-processor interrupts (IPI), is expensive, yielding overheads in the order of thousands of processor cycles. The overheads associated with expensive inter-core notification are neither addressed qualitatively nor simulated in the experiments presented [42]. Finally, scheduling work at the operating system level provides more flexibility than in hardware. For example, we show that exception-less system calls can be used to provide significant performance improvements when execution is restricted to *a single core*, which would be difficult to do with an OS transparent approach.

2.2.7 Optimizing Software Communication: IPC and System Calls

A set of operating system services that have been traditionally considered to be performance critical are the primitives used to communicate between protected domains. Example primitives are inter-process communication (IPC), remote procedure calls (RPC), system calls, shared memory, and signals.

In the context of micro-kernels, IPC implementations and their performance has been widely studied [26, 27, 48, 64, 84, 115]. Some of the optimizations used to reduce the cost of IPC include: using hardware registers for short messages, using hand-off scheduling techniques for synchronous, and fast, transfers of control, and using shared-memory in conjunction with polling.

Of particular interest to our work is the IPC primitive proposed by Gamsa et al. for the Tornado operating system, called Protected Procedure Call (PPC) [84]. The Tornado PPC was specifically designed to perform well on multi-processor systems, avoiding global locks or accessing shared data. They employ a “worker thread” model where control is transferred from the caller to one of the worker threads executing within the server.

A similar IPC mechanism, used to communicate between two user-level processes, is the user-level remote procedure call (URPC) mechanism introduced by Bershad et al. [27]. In URPC, the operating system provides shared memory that is used to communicate requests. In addition, URPC uses light-weight threads to poll the shared memory region and execute pending requests, consequently bypassing the operating system kernel in user-to-user communication.

As we describe in Chapter 4, the exception-less system call mechanism can be considered as

an asynchronous inter-domain messaging system which has been adapted specifically for system call communication. Conceptually, it shares elements, and was inspired by, the Tornado PPC and URPC mechanisms. Specifically, the use of light-weight threads to process messages from a separate domain is similar to both the above-mentioned projects and our exception-less system call mechanism.

A principle goal behind exception-less system calls is to reduce the costs associated with a specific type of communication: the communication between applications and operating systems. This communication has traditionally been encapsulated as a standardized interface known as system call. In the remainder of this section, we outline some techniques that aim to reduce the overhead of mode switching, which is one of the costs associated with system calls. As we show in Chapter 4, there are other overheads inherent to traditional system calls that are not addressed by these studies.

Specific to reducing the costs of system calls, which requires switching protection domains, are *multi-calls*. Multi-calls are used in both operating systems and paravirtualized hypervisors as a mechanism to address the high overhead of mode switching. Cassyopia is a compiler targeted at rewriting programs to collect many independent system calls, and submitting them as a single multi-call [157]. An interesting technique in Cassyopia is the concept of a *looped multi-call* where the result of one system call can be automatically fed as an argument to another system call in the same multi-call. In the context of hypervisors, both Xen and VMware currently support a special multi-call hypercall feature [18, 193].

Another strategy for reducing the number of domain crossings is to execute application code within the context of the kernel [29, 71, 72, 89, 153, 171, 178]. The challenge with executing user provided code in the kernel context are implications to security and reliability of the operating system. Several proposals restrict to use of extensions to specific type-safe languages that can be verified at run-time, other proposals use an in-kernel interpreter to execute extension code, while others use *sandboxing* through binary rewriting along with verifiers to guarantee the isolation of the extension.

Chapter 3

Software Pollute Buffer

It is well recognized that the least recently used (LRU) replacement algorithm can be ineffective for applications with large working sets or non-localized memory access patterns. Specifically, in processor caches, LRU can cause cache pollution by inserting non-reuseable elements into the cache while evicting reusable ones. In this chapter, we explore an operating system technique to improve the performance of applications that exhibit high miss rates of secondary processor caches.

A principal insight behind our technique is that, for certain applications, access patterns are distinct for different regions of an application's address space. In the case that one of the regions exhibits LRU unfriendly access patterns, there is potential for intra-application interference in the cache hierarchy where data of a low reuse region evicts data of a high reuse region. In this chapter we establish two properties of memory intensive workloads: (1) applications contain large-spanning virtual memory regions, each exhibiting a uniform memory access pattern, and (2) at least one of the regions does not temporally reuse cache lines.

The work presented in this chapter addresses secondary-level cache pollution resulting from *intra-application interference* through a dynamic operating system mechanism, called **ROCS**, requiring no change to underlying hardware and no change to applications. ROCS employs hardware performance counters available on commodity processors to characterize application cache behavior at run-time. Using this online profiling, cache unfriendly pages are dynamically mapped to a *pollute buffer* in the cache, eliminating competition between reusable and non-reusable cache lines. The operating system implements the pollute buffer through a page-coloring based technique, by dedicating a small slice of the last-level cache to store non-reusable pages. Measurements show that ROCS, implemented in the Linux 2.6.24 kernel and running on a 2.3GHz PowerPC 970FX, improves performance of memory-intensive SPEC CPU 2000 and NAS benchmarks by up to 34%, and 16% on average.

3.1 Introduction

Cache pollution can be defined as the displacement of a cache element by a less useful one. In the context of processor caches, cache pollution occurs whenever a non-reusable cache line is installed into a cache set, displacing a reusable cache line, where reusability is determined by the number of times a cache line is accessed after it is initially installed into the cache and before its eviction.

Modern processor caches are designed with the premise that recency ordering serves as good prediction for subsequent cache accesses. Hence, caches typically install all cache lines that are accessed by the application, expecting subsequent accesses to the same lines due to *temporal and spatial locality* in the application's access pattern. Moreover, hardware data prefetchers are widely used to populate caches by tracking sequential or striding access patterns to predict future accesses.

Both LRU caching and prefetching have been shown to be effective for the performance of many applications. As such, these techniques have been incorporated into processor design for decades. However, it has also been noted that these two techniques can perform poorly for some access patterns found in real workloads. In essence, the *mispredictions* that occur in LRU caching and prefetching are responsible for cache pollution, where lines are brought into the cache with the expectation of timely reuse, but which in fact are not accessed in the near future, replacing potentially more useful cache lines.

The computer architecture community has extensively studied the problem of cache pollution caused by both LRU placement and prefetching. Numerous enhancements to the memory hierarchy have been proposed and shown to be effective in mitigating the negative performance impact of cache pollution [68, 85, 99, 120, 149, 150, 154, 191]. Unfortunately, however, modern processors continue to be shipped with little or no tolerance to secondary-level[1] cache pollution. *The goal of the work presented in this chapter is to address secondary-level (L2, in this study) cache pollution caused by LRU placement and prefetching, providing a transparent, software-only solution.* We focus on the design of a run-time cache filtering technique at the operating system level that can be deployed on current processors.

The main insight this work builds upon is that coarse-grain (page-level) cache behavior is indicative of cache line behavior for the lines within the page, especially as it relates to pollution behavior. We show how it is possible to monitor and characterize cache pollution at page granularity using commodity hardware performance counters. With a full cache profile of an application's address space, we can identify per-page cache pollution effects from the observed miss rates.

We introduce the concept of a software-based *pollute buffer*, implemented in secondary-level caches for the purpose of hosting application data likely to cause pollution. In essence, the pollute buffer is used to reduce the *intra-application interference* observed in several memory intensive applications. We exploit the use of the pollute buffer in conjunction with online cache profiles to

[1] In this work, *secondary-level* caches refers to the levels of caches below the first level cache (L1), which have larger capacity and longer access latencies than the L1 cache.

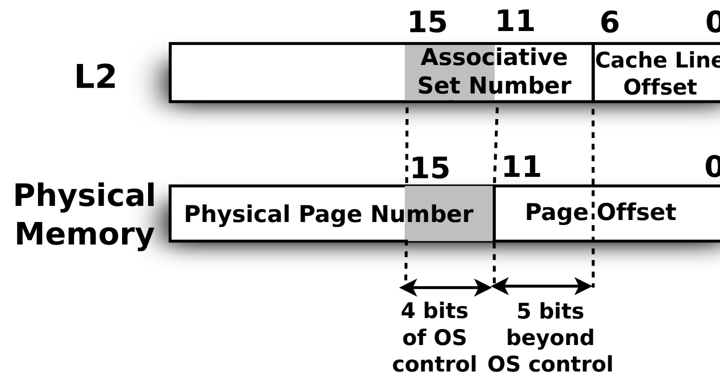


Figure 3.1: Cache indexing of physical addresses in the PowerPC 970FX processor, with 128B cache lines, L2 cache with 512 sets, and 4KB pages.

improve the performance of memory intensive applications.

We describe our implementation of a Run-time Operating system Cache-filtering Service (ROCS), in the context of the Linux kernel. Running on a real PowerPC 970FX processor, we evaluate its benefits, showing performance improvements of up to 34% on workloads from SPEC CPU 2000 and NAS, and an average of 16% on 7 memory intensive benchmarks from these suites. We also show that, in addition to reducing L2 cache miss rates of application data, mitigating cache pollution can benefit performance by reducing the L2 cache miss rate of performance critical meta-data, such as page-tables.

3.2 Background

In this section we provide a brief background on the two essential components used in this work: software cache partitioning and hardware performance counters. We leverage the concept of software cache partitioning to implement a software-based *pollute buffer* in the last-level cache. In addition, we make unconventional use of hardware performance counters to obtain online cache characterization of the target application's memory pages.

3.2.1 Software Cache Partitioning

An introduction to software cache partitioning was given in Section 2.2.5. For the convenience of the reader, we summarize the key concepts in this section, and, in addition, we illustrate, through a simple example, how the operating system can use page-coloring to create logical partitions in physically indexed caches.

Software partitioning of physically indexed processor caches (e.g., L2 and L3) is possible through operating system page-coloring [118, 125, 177, 202]. In physically indexed caches, physical addresses of data are used to map data into cache sets. The hashing function used for indexing into the cache must utilize enough bits from the address to be able to index anywhere into the cache. Due to the relatively large size of current caches, the number of bits required is larger then the num-

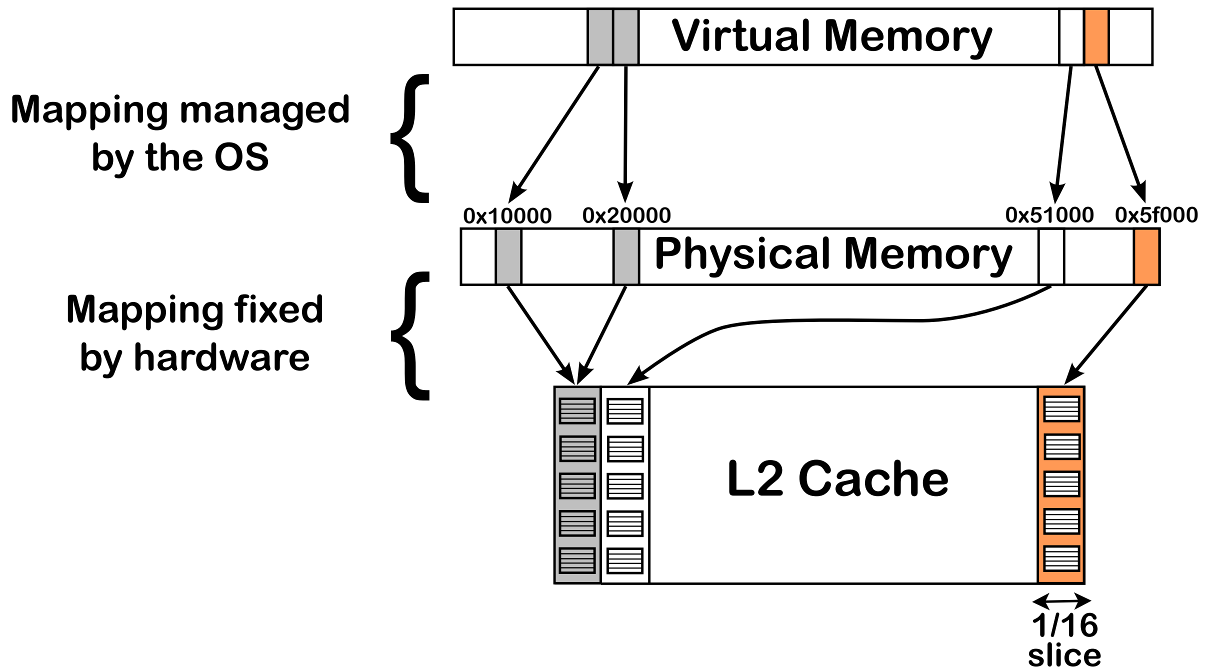


Figure 3.2: Example of L2 cache partitioning through operating system page coloring.

ber of page offset bits, as shown in Figure 3.1. As a consequence, the choice of virtual to physical mapping influences the specific cache sets which store application data.

Conceptually, the set of pages which share the indexing bits above the page offset form a congruence class. Each congruence class is mapped to a fixed partition of the cache. The number of congruence classes available is equal to 2^n, where n is the number of bits used for cache hashing above the page offset. In the case of the processor used in this study, the PowerPC 970FX, the L2 is organized into 512 sets, resulting in $\log_2(512) = 9$ bits used for indexing. The four most significant of those bits can be used to determine congruence classes. As a consequence, the operating system can control $2^4 = 16$ different cache congruence classes or *partitions*.

A concrete example of L2 cache partitioning through operating system level page coloring is shown in Figure 3.2. We depict 3 of the 16 possible partitions (for the case of the PowerPC processor used in this study) of the L2 cache. The left most partition is indexed by physical pages whose addresses are in congruence class 0 (i.e., page address modulo 2^{16} is equal to 0). This is the case for the two physical pages with address `0x10000` and `0x20000`. The next cache partition is indexed by physical pages with addresses in congruence class 1, which is the case for the page shown in the diagram with address `0x51000`. Finally, the right most cache partition in the diagram is indexed by pages with congruence class 15; one such page is depicted with address `0x5f000`.

Despite the fact that the indexing of physical addresses is fixed in hardware, the operating system can use the virtual memory system of modern processors to control where portions of application's address space map to, in the cache. As shown in Figure 3.2, two consecutive pages in the virtual address space can be controlled to map to the *same* cache partition (e.g., the two left-most pages of the virtual address space in the diagram). Alternative, two consecutive pages in the

virtual address space can be controlled to map to *different* cache partitions (e.g., the two right-most pages of the virtual address space in the diagram).

3.2.2 Hardware Performance Counters

Processor manufacturers have equipped modern processors with performance monitoring units (PMU). These are exposed to system software in the form of hardware performance counters (HPCs) and attendant control registers. The PMU can be programmed to count a wide range of micro-architectural events, including committed instructions, branch mispredictions, and L1/L2 hits and misses. Depending on the specific processor, PMU events that can be captured can number in the hundreds. However, the number of physical HPCs is much lower, typically less than 10, limiting the number of events that can be monitored concurrently.

HPCs can be read either through polling or through interrupts. For fine-grained monitoring, where performance characterization of a small time-slice or section of code is desired, the monitoring software can poll the content of the HPCs at the beginning and end of the slice. Coarse-grained monitoring, on the other hand, is done by programming the PMU to generate an interrupt when an HPC overflows. The operating system can then notify monitoring software, or simply accumulate the values.

Instruction sampling, a technique used by profiling software such as DCPI [6], Oprofile [2], and VTune [3], also uses hardware performance counters. For instruction sampling, the PMU is programmed with a sampling threshold and an interrupt is raised every time the threshold number of events of a specified type occur in the processor. Profiling software then attributes the event to the instruction of the current application program-counter, resets the HPC and continues to profile. After many samples, it is possible to determine the contribution of each instruction in the occurrence of the programmed event.

The wide-spread use of aggressive out-of-order processors has made interrupt-based instruction sampling less accurate. Since performance monitoring interrupts are *imprecise*, it is difficult to determine the exact instruction and/or address which triggered an event. This motivated ProfileMe, which attempts to provide accurate sampling by *marking* a single instruction in the pipeline and reporting events triggered by that instruction [59]. The PowerPC processor used in our work supports a similar mechanism, called *instruction marking*. Modern x86 architectures have also recently adopted precise marking of instructions throughout the pipeline; in the Intel architecture, the facility is called precise event-based sampling (PEBS) [183], and in the AMD architecture, the facility is called instruction based sampling (IBS) [66].

The biggest disadvantage of instruction marking is that of low recall: only a small subset of the instructions which cause an event of interest are profiled. This comes from the fact that only one instruction can be marked in the pipeline at a time. While the marked instruction is traversing the

[2] `http://oprofile.sf.net/`

[3] `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/`

pipeline, other instructions of interest may be concurrently executing and will pass undetected. Another contributor to low recall is the fact that marking occurs early in the pipeline, typically in the fetch unit. At that stage, it is not yet possible to determine if the marked instruction will cause any of the events of interest. In the case that it does not, the PMU must wait for the current instruction to commit before a next instruction can be marked.

3.3 Address-Space Cache Characterization

Our dynamic cache-filtering mechanism is based on run-time, page level cache characterization. Our system uses address-space cache profiles to identify memory pages that cause secondary-level cache pollution. In this section, we demonstrate how hardware performance counters can be used to build cache behavior profiles at page granularity. In addition, we present the characterization of 8 benchmarks from the SPEC CPU 2000 benchmark suite, providing insights for the creation of our software pollute buffer. We provide evidence that these workloads exhibit cache pollution that can be accurately identified at page granularity.

3.3.1 Exploiting Hardware Performance Counters

To obtain page-level L2 cache profiles of applications, we built a Linux kernel module which uses the PMU of the processor. In essence, the monitoring module identifies the data addresses of load requests that miss the L1 data cache, as well as the level of the cache hierarchy in which the data was found (either the L2 or main memory on our hardware).

The kernel module configures the PMU to mark instructions that access memory (load and store instructions) for monitoring, as described in Section 3.2.2. We specifically target loads that miss the L1 data cache (i.e., L2_HITS and L2_MISSES) so that a PMU interrupt is generated on every such event. For every PMU interrupt received, the module determines which HPC overflowed to determine from where the cache line is being fetched. It also reads the address provided by the *sampled-address data register* (SDAR) to obtain the virtual address of the cache access. In the PowerPC architecture, the SDAR provides the data address of the last marked load/store instruction, which in the case of our PMU configuration, will contain the loaded data address that caused the PMU interrupt.

Our module assembles page-level statistics on the addresses sampled, and creates a cache profile for each targeted address space. Each profile contains miss rates, as well as the proportion of accesses attributed to each virtual page in the address space. In essence, our profiling module performs accurate *data sampling* of L2 cache events. This technique is analogous to the widely used *instruction sampling*.

One issue with the approach described above is that aggressive interrupt handling affects the accuracy of the profiles generated since the PMU interrupt handler itself introduces L1 pollution. When handling an interrupt on an L1 miss, the data items used by the interrupt handler can cause

(hot) application data to be evicted to the L2. If the evicted line is still hot, it will immediately be fetched from the L2, adding artificial L2 hits for some pages in the profile.

To eliminate the effects of interrupt handling interference, we throttle the rate of interrupts so that hot cache lines evicted from the L1 by the interrupt handler have time to be brought back into the L1 by subsequent application accesses. We have empirically verified that for memory intensive SPEC CPU 2000 benchmarks, interrupt rates lower than 1 interrupt per 5K to 10K cycles cause minimal impact to the cache profile.

A second potential issue affecting the accuracy of data sampling based profiling is the imprecise nature of PMU interrupts. As discussed in Section 3.2.2, it is generally hard to determine the address of the load/store instruction which missed in the L1 cache. Therefore, we rely on instruction marking to accurately determine the data address of the L1 miss. Because instruction marking tracks a single instruction at a time and must track the complete life cycle of the instruction throughout the pipeline, from fetch to commit, concurrent instructions that access L2 can execute without being monitored. As a result, with instruction marking we have observed low recalls (as low as to 10%) in memory intensive workloads; meaning that sometimes we are only able to track 1 in 10 accesses to L2.

Fortunately, the profile we construct is statistical in nature and is used to compare the cache behavior of different pages, so the absolute values of cache accesses is not necessary. So the low recall of instruction marking is not an obstacle to construct our profiles, given that random samples are sufficient. Furthermore, as discussed above, to avoid the imprecisions in the profile due to L1 pollution, we target interrupt frequencies lower than once every 5 to 10K cycles. The 10% worst case recall rate that we observe is more than sufficient for our targeted interrupt frequency.

3.3.2 Empirical Simulation-based Validation

In this section, we present experimental results to show that the information collected using the hardware performance counters is sufficiently accurate to identify distinct access patterns within an application's address space. We present results from simulation-based experiments, using SPEC CPU 2000 workloads, focusing on their cache behavior at a per page granularity. We visually compare the address space profiles collected through simulation with profiles collected using hardware performance counters, as described in the previous section.

The simulation experiments were executed using the Simics full-system simulator [126]. We implemented a cache module that receives all cache accesses from the Simics simulator. With this module, we were able to characterize the application address space with respect to cache miss ratios. The parameters used in the simulated base-line cache are displayed in Table 3.1. These parameters were chosen to imitate the PowerPC 970FX processor, which is the processor we used as our experimentation platform to evaluate ROCS.

For the purpose of this study, we have not included timing information or detailed out-of-order processor execution. Our goal was only to observe access patterns present in workloads.

Cache line	size	128 B
L1 i-cache	size	64 KB
	associativity	2-way
L1 d-cache	size	32 KB
	associativity	2-way
L2	size	512 KB
	associativity	8-way

Table 3.1: Cache parameters used in base-line simulation experiments. These were chosen to mimic the PowerPC 970FX processor.

Although out-of-order execution can show changes in cache access patterns, particularly when introducing concurrency in access streams, it is unlikely that it will affect per page cache miss ratios in a significant way.

The workloads used in these experiments were benchmarks from the SPEC CPU 2000 suite. We chose benchmarks that exhibit high L2 miss ratios (i.e., applications that are memory-intense). For applications with low L2 miss ratio, our technique cannot provide any performance improvements, as these do not suffer adverse effects from cache pollution. The specific benchmarks chosen were: *ammp*, *art*, *mgrid*, and *swim*. Finally, for the simulated experiments, we skipped the first 1 billion instructions for application initialization and warm-up. Thereafter, we measured the subsequent 1 billion instructions.

Figures 3.3 to 3.6 show cache miss profiles of virtual address spaces the four applications. For each profile the x-axis represents a compaction of the virtual address space, and not the entire 32-bit address, for improved visualization. We sort virtual addresses of pages available in the corresponding trace, and renumber these sequentially. The y-axis displays the number of accesses that are hits, at the bottom, and misses, stacked on top.

In all figures we depict two graphs. The bottom-most graph corresponds to data collected during the execution of 1 billion instructions within the cache simulator, and should serve as a reference. The top-most graph corresponds to the data collected by our kernel module using the hardware monitoring unit, when monitoring during *4 seconds* of execution of the application.

The graphs indicate that sampled hardware monitoring provides a characterization of the address space similar that of the simulation-based results. There are two notable differences between the two graphs: the range of virtual addresses, represented by the x-axis of the graphs, and the number of accesses observed by each page, represented by the y-axis. The reason behind the variations in both of the axes stems from the fact that with sampling some information is lost. In the case of the differences between the range of virtual addresses, the sampled profile can miss pages that are accessed only a few times. As a result, the profiles using sampled data contain less virtual pages. This does not pose a problem since pages that are accessed seldomly are unlikely to be a significant source of cache pollution. With respect to the total number of accesses, the sampled monitoring, by definition, does not capture all accesses to L2. For this reason, we observe between

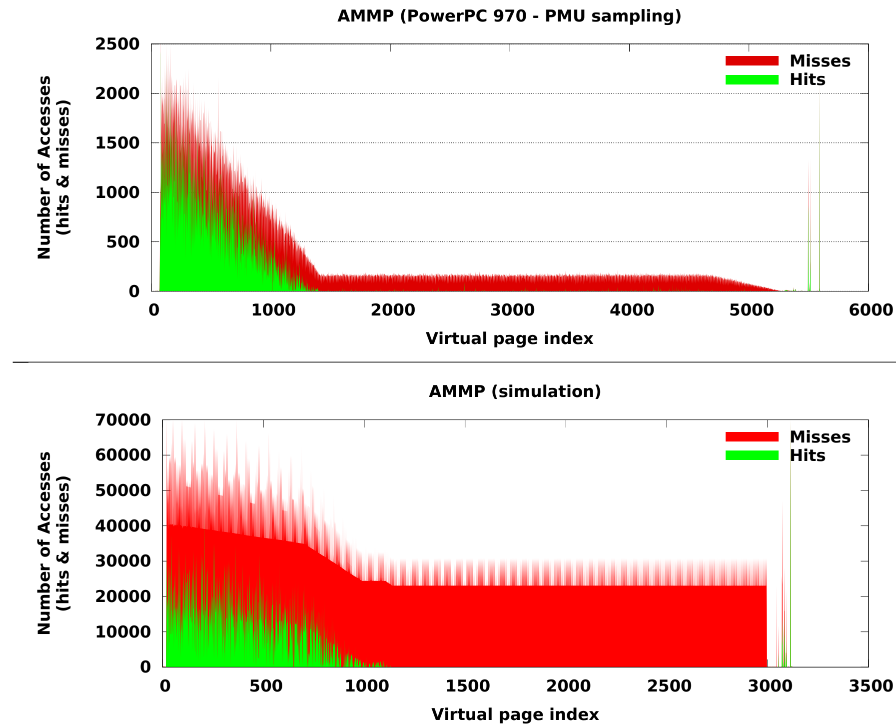


Figure 3.3: Virtual page cache access characterization of AMMP. The top graph shows 4 seconds worth of sampling on hardware, and the bottom graph shows simulation of 1 billion instructions.

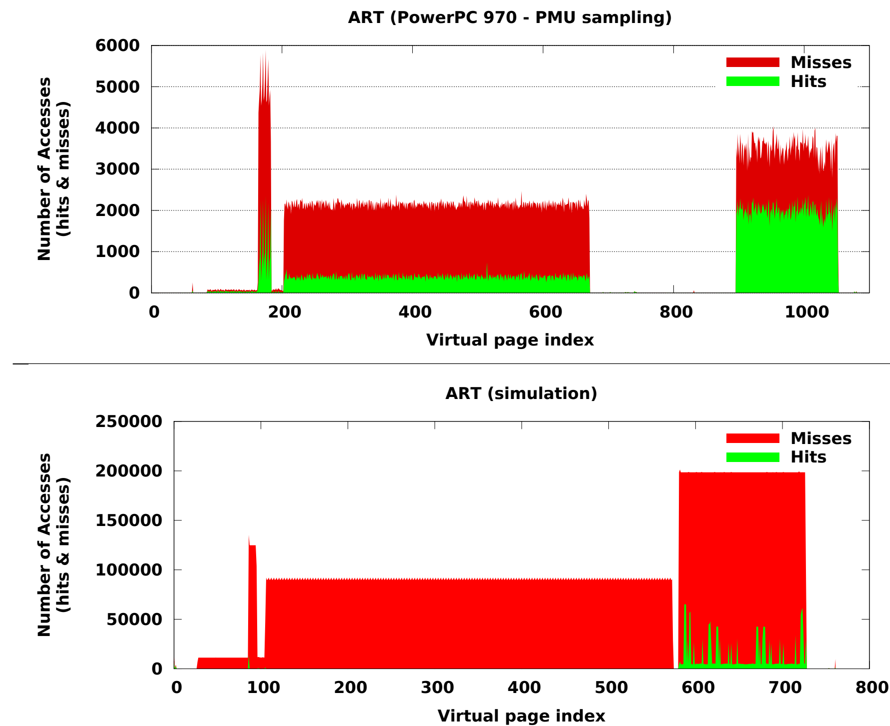


Figure 3.4: Virtual page cache access characterization of ART. The top graph shows 4 seconds worth of sampling on hardware, and the bottom graph shows simulation of 1 billion instructions.

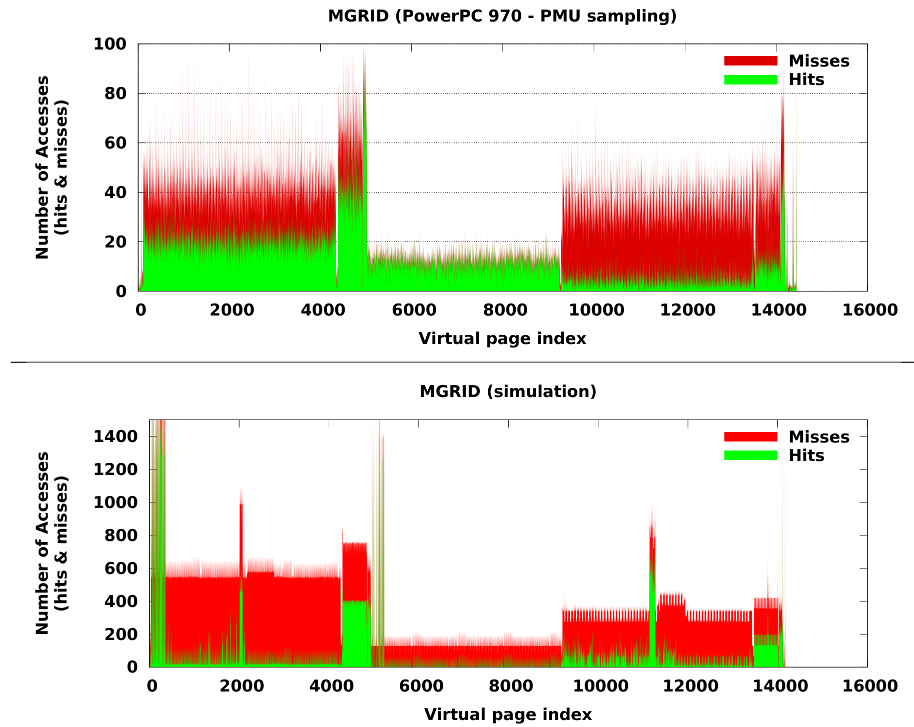


Figure 3.5: Virtual page cache access characterization of MGRID. The top graph shows 4 seconds worth of sampling on hardware, and the bottom graph shows simulation of 1 billion instructions.

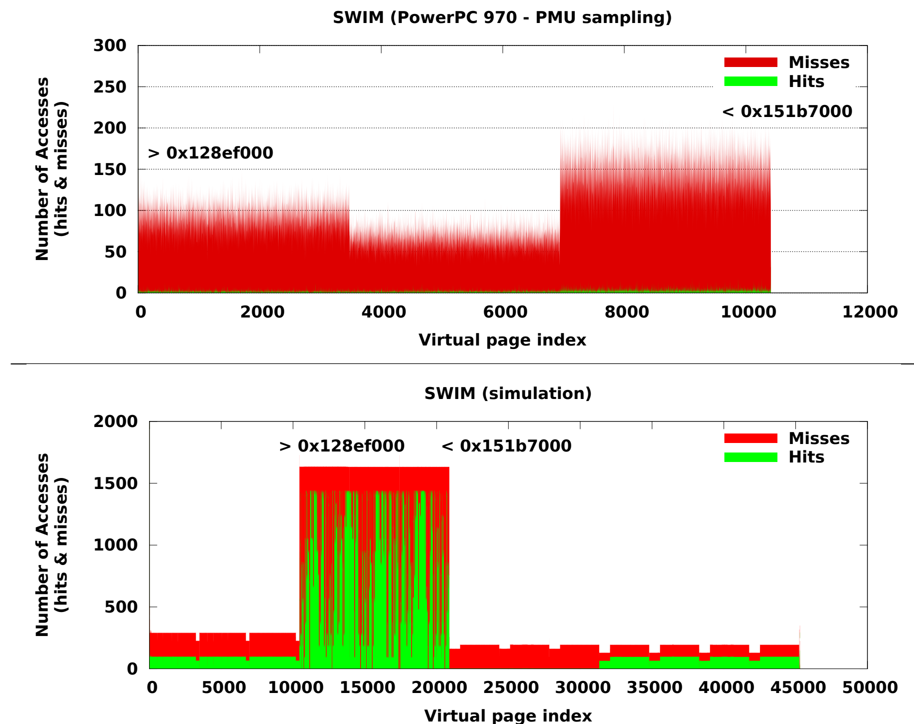


Figure 3.6: Virtual page cache access characterization of SWIM. The top graph shows 4 seconds worth of sampling on hardware, and the bottom graph shows simulation of 1 billion instructions.

10x to 30x less points when sampling than the actual number of cache accesses. However, in these profiles we are not interested in *total* number of accesses to each page, but rather, that the per page relative *miss ratio* is preserved.

There is one case, *swim*, where the sampled monitoring graphs exhibit different shapes than the simulation-based one. For *swim* (Figure 3.6), the hardware sampling misses a large number of pages that have relatively low number of accesses (pages with less than 400 accesses in the simulation-based plot). However, we have annotated the graph with the actual virutal addresses that delimit the region of memory with high miss ratios. Fundamentally, the region of memory identified as cache polluting with continuous monitoring is also correctly identified with sampled monitoring.

3.3.3 Page-Level Cache Behavior

Page-level profiling is oblivious to potential miss-rate[4] variances of cache-lines within a page. Nonetheless, we demonstrate that profiling at page granularity provides insights on the application cache behavior at run-time. We show, in the next section, how page-level profiling can help identify pages that cause cache pollution. In addition, we analyze the cache profile of the *art* benchmark, as a case study, showing how the profile relates to *art*'s source code.

Classifying Pollution

An essential function of our cache filtering system is to classify pages with low and high reuse of cache-lines, such that it is possible to determine which pages should have restricted cache access. Previous work attempted to classify cache pollution, at cache line granularity, based on single-use or zero reuse of cache lines [149, 154].

Given the lack of fine-grained monitoring in commodity hardware, we make the simplifying assumption that the L2 miss rate of a page directly correlates to how much it pollutes the cache. The empirical justification is that pages with high miss rates experience little benefit from being cached, since each miss results in an eviction of a potentially useful cache-line. That is, pages with high miss rates cause high rates of cache-line evictions.

The per-page miss rate can be viewed as an inverse measure of the probability of reusing a cache block of the page after its insertion in the cache. We show in Section 3.6 that pages with low probability of reuse (1) have limited benefit from caching and (2) negatively impact pages with high probability of reuse. Our approach uses this assumption to constrict the caching of pages with low reuse probability, consequently increasing the effective cache space for pages with higher probability of reuse.

Figure 3.7 shows the distribution of per-page miss-rates for 8 memory intensive workloads from the SPEC CPU 2000 benchmark suite over the entire execution of the application. The graphs

[4] In this work, L2 miss rate is defined as the number of L2 misses divided by all L2 requests.

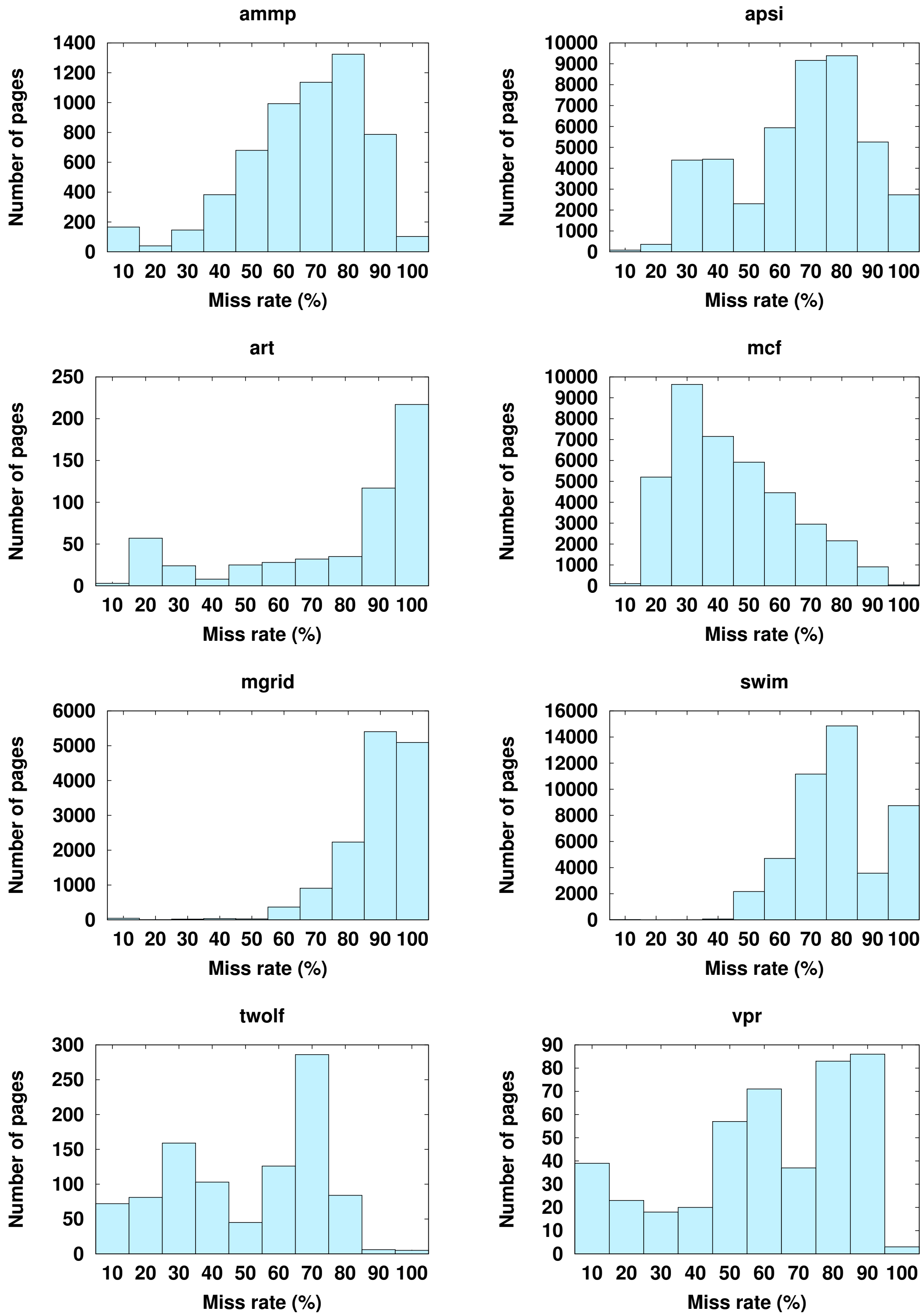


Figure 3.7: Page-level L2 cache miss rate characterization. The histograms show per-page distribution of miss rates.

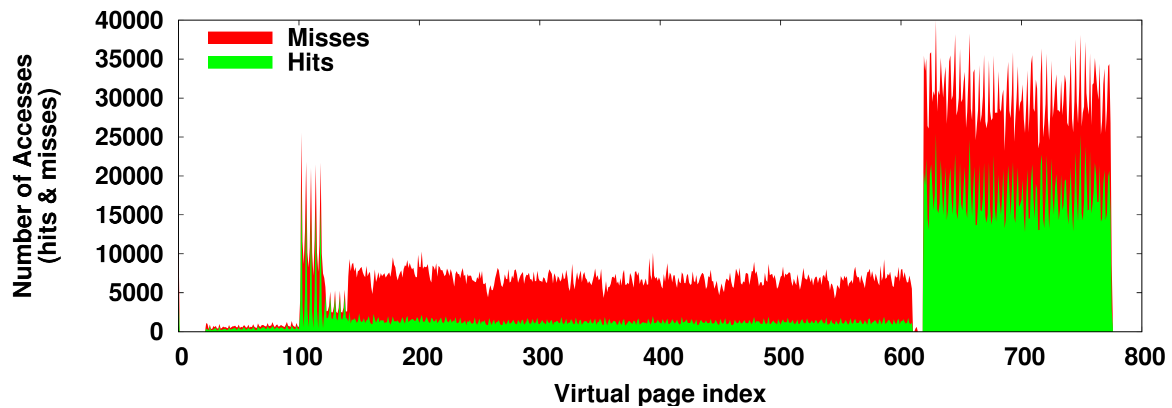


Figure 3.8: Page-level L2 cache miss rate characterization for *art*. The histogram shows a compact view of the address space, each bar representing accesses (hits and misses) to a page.

show that it is possible to identify a significant number of pages that exhibit high miss rates, and therefore, are likely to cause pollution in the cache. From the graphs shown, the only benchmark which does not show a large proportion of pages with high miss rates is *mcf*.

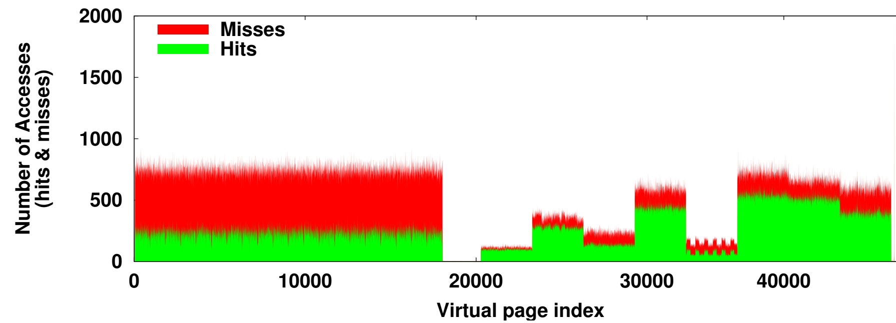
Case Study: *art*

We show that for some workloads cache miss behavior can be characterized at an even coarser granularity than pages, using the *art* benchmark. Figure 3.8 shows an example of the collected cache profile for the entire execution of *art*, depicting a compact view of its address space. In the profile, both the total number of accesses and the miss rates are shown for each page.

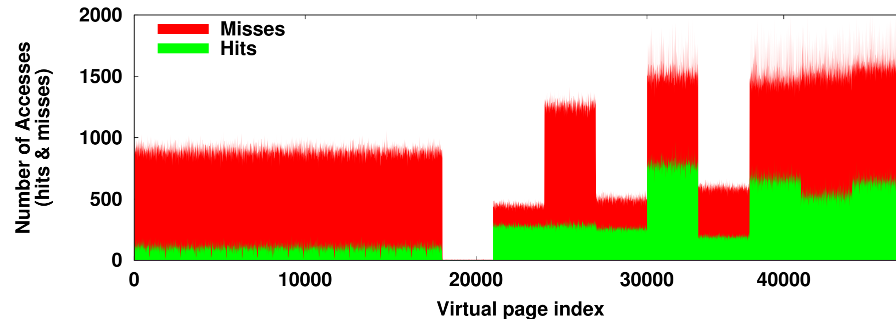
Art implements a neural network for image recognition. The significant data structures of *art* are comprised of three 2-dimensional arrays: f1_layer, an array of neurons, and; tds and bus, arrays of weights. In the profile shown, there are two large memory regions with distinct L2 cache behavior. The contiguous memory region to the left (pages 100 to 600) contains the two arrays: bus and tds. Accesses to these two arrays correspond to 39% of all accesses to L2 and their pages obtain an 81% miss rate in the L2. The rightmost memory region (pages 620 to 780) contains the f1_layer array. This regions corresponds to 56% of all L2 accesses and has an average miss rate of 42%.

This profile shows that the leftmost memory region (bus and tds) does not benefit significantly from the L2 cache. On the other hand, the benefit from caching the rightmost memory region is visibly higher. As we will show in our evaluation, limiting the L2 cache space of bus and tds arrays improves the cache hit rate of f1_layer by preventing L2 pollution and consequently improving the overall L2 hit rate and application performance.

This example provides an important insight: many applications contain distinct memory regions, each with its own uniform cache behavior. These regions are sufficiently coarse-grain so that page-level cache management is applicable. Coarse-grain tracking and management of memory has been observed before in the literature for improving snoop-coherence and data prefetching [52, 206]. This work confirms that the same observation applies to caching behavior.



a) Profile with prefetching enabled



b) Profile with no prefetching

Figure 3.9: Page-level L2 cache miss rate characterization for *wupwise*, with and without prefetching.

Prefetching Interference

Modern high performance processors employ data prefetchers in order to minimize access latency. Due to their significant performance benefits, data prefetchers have been implemented in multiple levels of the memory hierarchy. The L1 data prefetcher implemented in the PowerPC processor, although quite simple, is able to significantly improve performance of programs with sequential memory references.

For characterizing cache behavior, and specifically for classifying cache pollution of pages based on cache miss rates, the prefetcher poses two problems. The first problem is the existence of *invisible lines*. Cache lines from pages that are prefetched into L1, although occupying entries of the L2, are not fully counted in the profile, because the profile is based on the source resolution of demand loads that miss L1, leading to a perceived lower occupancy in the L2.

The second problem is that of *artificial hits*. An artificial cache hit occurs when prefetching from memory is not timely enough to bring the line to L1, but timely enough for L2 insertion. As a consequence, an L1 miss occurs, which is satisfied from the L2. This causes L2 cache hits to be incorporated into the profile even though these hits are *not* a result of cache line reuse, but a result of prefetching. Effectively, *artificial hits* make pages appear to be more reusable than they are and, therefore, they appear to benefit from caching.

Ironically, the opposite conclusion should be drawn from highly prefetchable pages. For pages that exhibit hits from prefetching, caching becomes less important for hiding memory access la-

tency. For overall performance, it may be better to give pages that are not prefetchable higher priority in the cache. In addition, prefetching can bring useless lines into the cache when prefetch predictions are too aggressive, thus causing pollution. The possibility of pollution is another reason why prefetchable pages should have restricted cache access. Unfortunately, the hardware monitoring unit available in the processor we used (PowerPC 970) does not provide prefetch information about specific memory accesses, thus making it difficult to generate profiles containing pretech behavior of memory ranges or pages.

To overcome these issues, instead, we disable the hardware data prefetcher while generating cache profiles, but enable it for the remainder of the application. To illustrate both problems mentioned above, Figure 3.9 shows the memory profile of the *wupwise* benchmark with and without prefetching. The eight rightmost memory regions starting at virtual page index 20000 have significantly different characteristics depending on whether prefetching is enabled or not; prefetching reduces the perceived occupancy in L2 (*invisible lines*), and increases perceived reuse probability (*artificial hits*).

3.4 Software-Based Cache Pollute Buffer

The insights from the previous section motivate cache management at memory page granularity. For this purpose, we have designed a software-based cache *pollute buffer*. The pollute buffer provides a mechanism to restrict specific memory pages, deemed to pollute the cache, to a small partition of the cache. It is meant to serve as a staging space for cache lines that exhibit bursty or no reuse before eviction. By restricting cache unfriendly pages to the pollute buffer, we eliminate competition between pages that pollute the cache and pages that benefit from caching.

In our system, the pollute buffer is implemented with software-based cache partitioning. We do so by dedicating a single *partition* of the L2 to act as the pollute buffer. Figure 3.10 illustrates the design of the pollute buffer using page-coloring. As described in Section 2.2.5, this is possible by allocating physical pages for polluting virtual pages that map to a specific section of the cache, whose cache indexing bits are in the same congruence class.

An inherent property of the pollute buffer is that, since it uses a partition of the last-level cache, it is amenable to (1) errors in the classification of pages with respect to pollution, and (2) variances in the miss rate of individual cache lines of a pollute page. The pollute buffer, although small in size, continues to allow hits on frequently accessed cache lines since the LRU replacement policy remains unchanged. After all, the pollute buffer is part of the last-level cache.

In order to manage cache pollution at run-time, our system requires moving application pages from one cache partition to another (from the non-pollute part of the cache to the pollute buffer, or vice versa). To perform this task, we must *copy* the content of the virtual page from the old physical page to a newly allocated physical page that maps to the target partition of the cache. This involves (1) allocating a new empty page that maps to the desired partition, (2) removing the appropriate

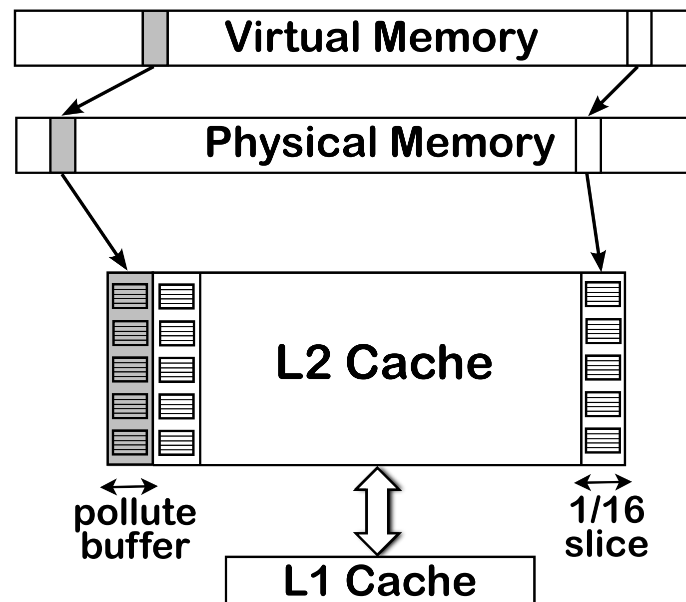


Figure 3.10: Representation of a *software pollute buffer*. The software pollute buffer is implemented by dedicating a partition of a secondary level cache to host lines from pages that cause cache pollution. To implement the pollute buffer, we exploit a well-known operating system technique called page coloring. At a high level, the operating system can map application virtual pages (top box) to a selected set of physical pages. These physical pages are selected based on their address so that, according to the indexing function of the secondary cache, the content of the pages will occupy a fixed, and small, partition of the cache.

page-table entry in the application page-table, potentially flushing a TLB entry, (3) performing a physical page copy, and (4) reinserting the page-table entry, with the physical address of the new page.

3.4.1 Kernel Page Allocator

For our implementation, we modified the Linux kernel page allocator to efficiently allocate physical pages so that they map to specific partitions of the cache. Default Linux relies on two structures to manage free pages for allocation. Each processor contains a local list of recently freed (hot) pages for fast allocation. A global structure, managed through the buddy allocator algorithm, contains the majority of free pages [109]. The buddy structure is organized as a binary tree that clusters contiguous physical pages hierarchically. The leaf nodes (0-order) contain single pages, the next level (1-order) contains clusters of 2 physically contiguous pages, and so on. This organization allows for fast allocation of physically contiguous pages, which are needed by devices that do not support virtual memory.

Our modified Linux splits both the CPU and buddy allocator lists into 16 lists; one list for each partition of the L2. When populating the list with a new free page, the physical address of the page is used to determine the correct list to use. To satisfy new page allocation requests that map to a specific cache partition, the allocation can be quickly serviced by removing a page from the

appropriate free-list. In cases where the allocation specifies multiple allowed partitions, round-robin is used between the lists, emulating bin-hopping [104].

3.5 Run-Time OS Cache-Filtering Service

In the previous two sections, we presented page granularity cache characterization and the concept of a software-based cache pollute buffer. With these two constructs, we now describe ROCS, our implementation of a run-time operating system cache-filtering service. We show how online memory page cache profiles can be collected and used to determine which application pages should be mapped to the pollute buffer.

3.5.1 Online Profiling

In Section 3.3 we presented a collection of address space cache profiles. The profiles shown were gathered from *complete* execution runs. Unfortunately, this is impractical to do for run-time software cache management, as the overhead is prohibitively high: profiling involves recording L1 misses through an operating system interrupt handler where each interrupt entails a complete pipeline flush, interrupt delivery, fetch of interrupt handling code and execution of the handler itself.

Figure 3.11 shows the overhead, in terms of execution time slowdown, of *art* with varying interrupt frequencies. As discussed in Section 3.3.1, unrestricted monitoring of L1 misses distorts profiling as the interrupt handler evicts application data from L1. Fortunately, interrupting every 5 to 10 thousand cycles has a two-fold benefit: more precise L2 characterization (as explained in Section 3.3.1) and significantly lower overhead. Unfortunately, overheads of 15-65% are still prohibitive for online cache reconfiguration.

To reduce overhead, ROCS uses phase-based sampling. Application cache profiles are gathered for a short period of application execution. During each period, we profile with the smallest threshold that yields acceptable accuracy (5K cycles). We use this sample profile as a representation of the current application phase. Further samples are taken when a *coarse-grain* phase change is detected. Phase changes can be cheaply monitored with hardware performance counters, by measuring, for example, IPC or the L2 misses per kilo instructions (MPKI) of the application [175]. Since only coarse-grain phases are tracked, the IPC may be computed every 1 billion cycles, which incurs negligible overhead (one interrupt per billion cycles).

It is important to note that ROCS, as an operating system component, is able to solely target processes or threads that exhibit high L2 miss rate. When non-targeted threads are scheduled, profiling can be disabled or restricted to monitoring L2 miss rate. In this way, no overhead is observed for application threads exhibiting low L2 miss rates.

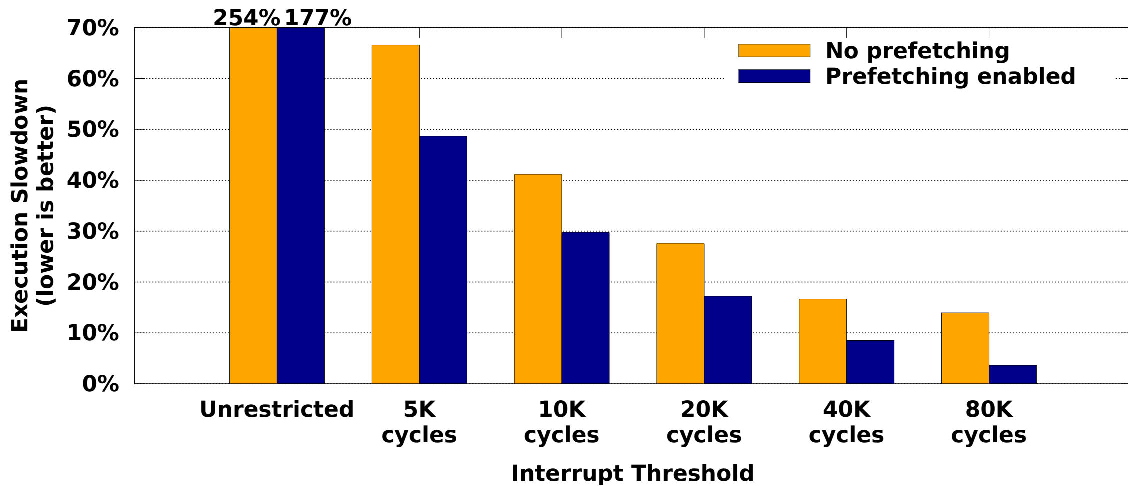


Figure 3.11: Overhead sensitivity of monitoring *art*. The graph shows the slowdown of execution with different interrupt thresholds when profiling cache accesses. Light colored bars show the slowdown of execution when the hardware prefetcher is disabled.

3.5.2 Dynamic Page-Level Cache Filtering

Given a page-level cache profile, a fundamental challenge is to identify which pages should be restricted to the pollute buffer in order to improve application performance. Addressing this challenge requires predicting the effects of restricting pages to the pollute buffer. This is analogous to predicting the effect of partitioning shared caches between different applications, with the difference being that we are potentially partitioning the cache between different memory pages of the *same* application.

Previous work on online prediction of the performance impact of shared cache partitioning between different applications required information on per-application miss-rates as a function of cache size (e.g., miss-rate curves) [148, 154, 185]. In our context, this would require obtaining per memory-region utility information, which is too expensive to compute in software at run-time – Berg et al. report a 40% average run-time overhead with sparse sampling [25], but their technique is oblivious to memory regions.

Given the high overhead and complexity of analytically predicting interference in the cache, we instead employ an empirical search algorithm. From the cache profile described, we construct a miss-rate stack, containing all pages in the monitored address space, ordered by miss rate (highest miss-rate at the top). We then proceed to monitor the improvement by mapping different numbers of pages from the stack to the pollute partition.

The pseudo-code of the search algorithm is listed in Algorithm 1. Starting at the top of the miss rate stack, where the pages are most likely to be cache polluters, a number of pages are remapped from their original cache partition to the dedicated pollute buffer slice. We use the hardware performance counters to evaluate the performance (IPC) of this mapping. Next, a subsequent number of pages are remapped to the pollute buffer, adding to the pages already mapped to the pollute

Algorithm 1 FindPollutePages: returns the number of pages from $mrateStack$ mapped to the pollute buffer.

```
procedure FINDPOLLUTEPAGES(mrateStack, stepSize)
    index ← mrateStack.size();
    while index > minPages do
        MAPTOPOLLUTEBUFFER(mrateStack, index, stepSize);
        performance ←MONITORPERFORMANCE();
        if performance > best then
            best ← performance;
            pollute_index ← index;
        end if
        index ← index − stepSize;
    end while
    UNMAPFROMPOLLUTEBUFFER(mrateStack, index,
                                  pollute_index − index);
    return mrateStack.size() − pollute_index;
end procedure
```

buffer. This iterative algorithm continues until a minimal set of pages is reached (less than $stepSize$ pages are left unmapped). The best configuration is recorded, and the stack is traversed upwards, restoring the excess pages of the pollute buffer to the non-pollute slices of the cache, if necessary.

Despite the simplicity of the algorithm, we found that the algorithm only takes 3 billion cycles, on average, and 7 billion, in the worst case, for SPEC CPU 2000 benchmarks. On our evaluation platform, with a 2.3GHz processor, this is equivalent to an average of 1.4 seconds, and a worst case of 3.1 seconds (as we will show in Section 3.6, this corresponds to less than 3% of the execution time of SPEC CPU 2000 benchmarks). It is important to note that not all this time corresponds to overhead. During the execution of the algorithm the application continues to make progress, although at potentially reduced efficiency due to suboptimal mapping of pages and the fact that pages are copied physically copied at the beginning of each iteration of algorithm. In the end, we expect that the overhead is much lower than the total time taken by the algorithm.

Furthermore, this search is significantly faster than published approaches for deriving L2 cache miss-rate curves in software [24, 63, 174]. In addition, we show in the next section, that the overhead of searching for a good pollute mapping incurs, in most cases, less overhead then profiling the application address space.

3.6 Evaluation

Table 3.2 lists the relevant architectural parameters of our evaluation platform. The system under test is a PowerMac G5, with 2 PowerPC 970FX processor chips, clocked at 2.3GHz, built on a 90nm process. For all cases in our evaluation, we restricted application execution, monitoring and page remapping to a single processor, disabling the second CPU in the operating system. Baseline results were obtained using the Linux kernel version 2.6.24. ROCS was developed using the same

Component	Specification
Issue width	8 units (2 FXU, 2 FPU, 2 LSU, 1 BRU, 1 CRU)
Reorder Buffer	100 entries (20 groups of 5 instructions)
Cache line	128 B for all caches
L1 i-cache	64 KB, direct-mapped, 1 cycle latency
L1 d-cache	32 KB, 2-way, 2 cycle FXU latency, 4 cycle FPU latency
L2 cache	512 KB, 8-way, 12 cycle latency
Memory	2GB, 4KB pages, 300 cycle latency (avg.)

Table 3.2: Characteristics of the 2.3GHz PowerPC 970FX.

Benchmark	Exec. time	Instrs.	IPC	L2 MPKI	L2 Miss Rate
ammp	9m00s	365B	0.30	7.5	52%
apsi	5m29s	334B	0.45	6.5	61%
art	3m10s	44B	0.10	69.0	75%
mcf	8m23s	51B	0.05	68.3	54%
mgrid	2m43s	255B	0.70	3.0	25%
swim	18m33s	262B	0.10	22.7	75%
twolf	9m11s	261B	0.22	9.7	35%
vpr	2m10s	96B	0.33	5.9	25%
CG	22m42s	137B	0.16	42.1	59%

Table 3.3: Benchmark characteristics. The 8 top-most rows corresponds to benchmarks from the SPEC CPU 2000 benchmark suite, and the bottom-most row (*CG*) is from the NAS-serial benchmark suite.

Linux kernel version.

We evaluated ROCS using SPEC CPU 2000[5] and NAS-serial [15] (serial version of NAS 3.3) benchmark suites. For SPEC CPU 2000, the reference inputs were used, and "Class B" inputs were used for NAS-serial. Table 3.3 lists the benchmarks from these suites that exhibit L2 miss rates greater than 25% on our platform, along with the most relevant characteristics collected using hardware performance counters. All other benchmarks from the suites with less than 25% miss rate displayed far lower misses per kilo instructions (MPKI). We did not consider applications with low L2 miss rates, since our technique is targeted at workloads that exhibit L2 cache pollution. Recall that ROCS is able to identify applications with low L2 miss rates while incurring negligible overhead (see Section 3.5.1). For the NAS-serial benchmark suite, the only benchmark that displayed a L2 miss rate higher than 25% was CG.

[5] At the time this portion of the thesis was done, SPEC CPU 2006 was not easily available to us.

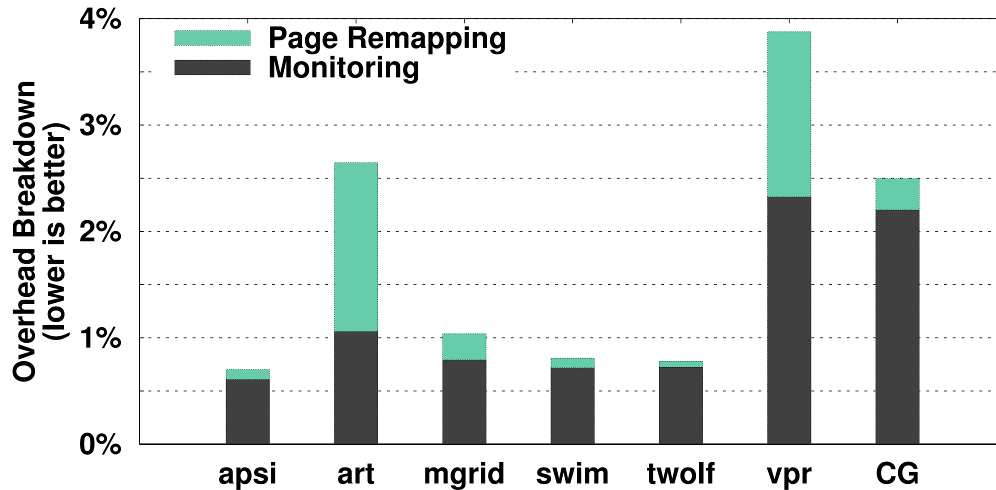


Figure 3.12: Run-time overhead breakdown of ROCS.

The size of the pollute buffer used in all experiments was 1/16th of the L2 cache; in our case, 32KB. All benchmarks were compiled for a 64-bit environment. We always present the average results obtained from three consecutive complete runs. An initial (discarded) run was used to ensure that all necessary files and binaries were resident in memory.

We excluded the *ammp* and *mcf* benchmarks from further performance analysis, as these benchmarks showed only around 1% improvement with ROCS. For the *mcf* benchmark, although it exhibits a high L2 MPKI (68.3) and a high L2 miss rate (54%), the misses are not concentrated around a cluster of pages of its address space, as show in Figure 3.7. While several of the benchmarks profiled have a significant number of pages with high miss rates (e.g., over 70% of accesses are misses), most of the pages of *mcf* have between 30% and 70% miss rate. For this reason, restricting any set of pages from *mcf* to use a small portion of the cache has only a marginal effect on performance.

The *ammp* benchmark, on the other hand, displays a different characteristic that reduces the performance potential when applying the on-line use of our pollute buffer. *Ammp* has multiple short phases (some with 1 billion instructions), making the overhead of each profiling phase higher than with other applications with longer program phases. Although ROCS is able to recoup the costs of on-line profiling by improving the IPC of *ammp* during the unmonitored portion of execution, the overall impact on performance is negligible.

3.6.1 Overhead

Figure 3.12 depicts the run-time overhead of ROCS, split into two components: monitoring and page remapping. We obtained these two overhead components by executing each application with three different configurations: a baseline with our system disabled, address space monitoring alone, and monitoring plus the page remapping algorithm with the caveat that physical pages were copied back to their original configuration after the last iteration of the algorithm. This last

configuration represents the worst case overhead if the use of the pollute buffer is found to be ineffective. The difference in performance between each configuration quantified the two contributions to overhead.

There are two main reasons behind the large variance in overheads between different applications. First, the different duration of the detected phases for each application leads to different overall overhead. Since monitoring and page remapping occur once per phase, applications with long phases amortize the overhead through longer periods of execution. Most of these benchmarks exhibit stable IPC after initialization (when monitoring IPC at 1 billion cycle granularity), ROCS initiates only 1 or 2 monitoring phases. So, in general, applications with longer execution lengths (listed in Table 3.3) observe less overall overhead.

A second reason for the variations relates to the size of the working set of the application. During the monitoring phase, we attempt to obtain a minimum number of samples per detected application page (in our experiments, we aimed at a minimum of 20 samples per page). Applications with larger working set sizes require longer periods of monitoring in order to meet this requirement.

It is also interesting to note that the overhead caused by monitoring overshadows the overhead due to remapping when application address spaces become large, despite the increase in pages remapped (see Table 3.4). Fortunately, applications that consume many pages typically also run for longer periods of time in stable phases in order to consume their entire data set. Consequently, we see an overall average overhead of 1.6%, with 3.8% being the worst case. As we show in the next section, this overhead is more than recovered by the performance improvements obtained through our technique.

3.6.2 Performance Results

Figure 3.13 shows the run-time speedup of three different cache filtering schemes. In all three schemes, the page miss rate stack was collected at run-time, after the first 4 billion cycles in order to avoid application initialization. *Best Offline* consists of a static exhaustive search for optimal stack values (number of pollute pages). This involves running the application multiple times, varying the number of pages to remap to the pollute buffer. The *ROCS* system incorporates the dynamic search algorithm as described in Section 3.5.2. In *ROCS*, the hardware data prefetcher is disabled while monitoring the application for its miss rate stack, but is enabled otherwise. Finally, we also show the performance of *ROCS* given a miss-rate stack generated with the hardware prefetcher enabled.

The average improvement of ROCS over Linux for the 7 benchmarks is 16.6%. The largest performance win of 34.2%, comes from *swim*. In all cases, we see that ROCS is able to approach the performance of optimal offline search. The worst case occurs with *apsi* where ROCS achieves 2.1% less speedup than the offline search.

The MPKI reductions of the benchmarks running under ROCS are shown in Figure 3.14, and

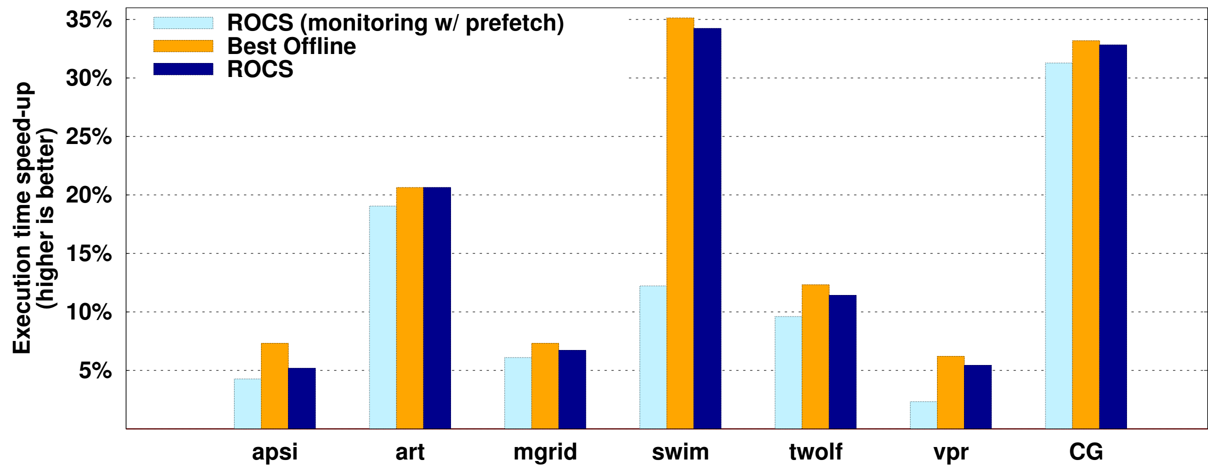


Figure 3.13: Performance improvement of ROCS over default Linux. The left-most bar for each benchmark corresponds to the improvement of execution with ROCS, but with the prefetcher enabled during cache monitoring. The middle bar

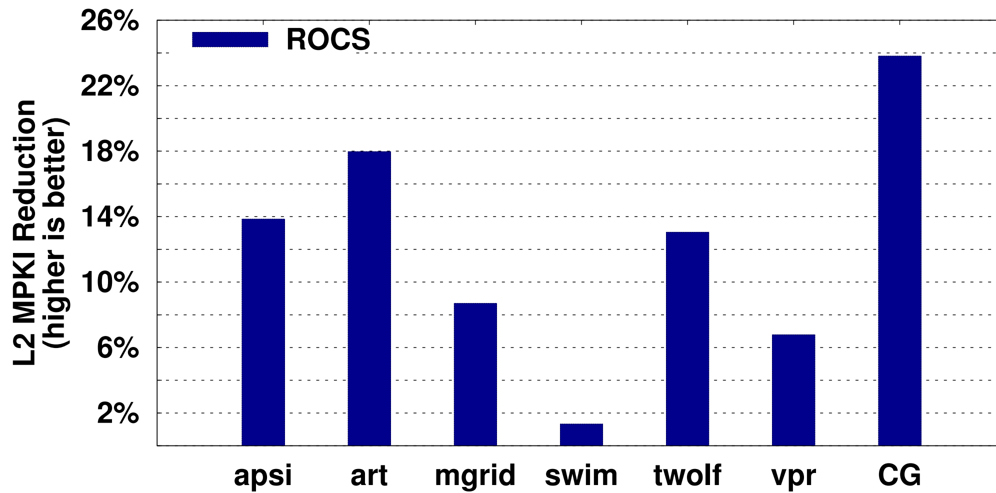


Figure 3.14: MPKI reduction with ROCS over a default Linux.

average 12.2%. For the most part, the MPKI improvements correlate with the performance improvements. The glaring exception is *swim*, which we analyze separately in Section 3.6.4.

The number of pages chosen as polluters by ROCS, and remapped to the pollute buffer is shown in Table 3.4. It is interesting to note that, with the exception of *swim*, there is a correlation between the fraction of pollute pages chosen by ROCS and the profile information shown in Figure 3.7. Applications that were shown to contain a higher fraction of pages with high miss rate obtained their best improvement by classifying a higher fraction of pages as cache polluters. This correlation corroborates our initial assumption that the degree of cache pollution, at the page-level, is directly related to its observed miss rate.

To further analyze our results, we discuss two specific cases in greater detail: *art* and *swim*.

Benchmark	Number of Pages	Number of Pollute Pages (% of all)
apsi	44159	2676 (6%)
art	865	607 (70%)
mgrid	14433	3157 (21.8%)
swim	45490	14335 (31.5%)
twolf	1530	181 (11.8%)
vpr	812	183 (22%)
CG	40818	7026 (17.2%)

Table 3.4: Classification of pollute pages.

3.6.3 Case study: *art*

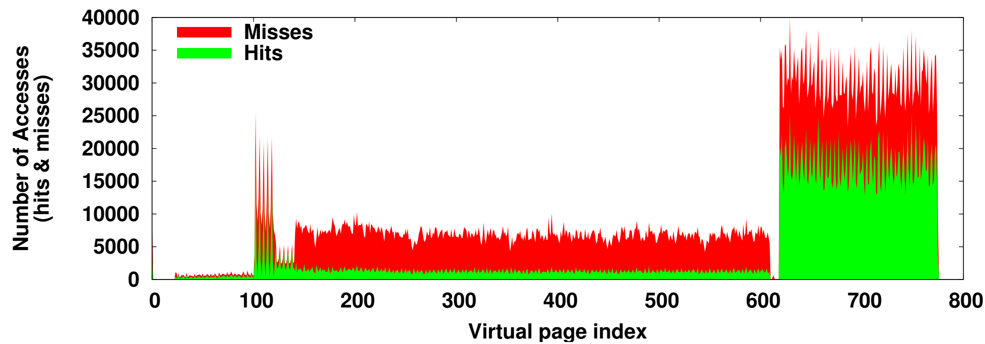
Figure 3.15 shows two cache profiles of *art*'s address space: on the left (a) we replicate the image from Section 3.3.3 containing the characterization of *art* on default Linux, and on the right (b) we show the profile of *art* on ROCS. As discussed, the leftmost memory region, with visibly high miss rates, contains two 2-dimensional arrays, bus and tds arrays. The rightmost region contains a single 2-dimensional array, f1_layer.

ROCS chooses to predominantly classify pages from the leftmost memory region as polluters, mapping them to the pollute buffer. The effects on the L2 access pattern can be seen on the graph to the right (b). With less competition from polluting pages, the f1_layer array sees a 16% reduction in its L2 miss rate (from 42% to 35%). In addition, a decrease of 6.7% in the *total* number of accesses to this region is visible in the graph. This decrease comes mainly from the L1 data prefetcher; with the reduced L2 miss rate, data prefetching becomes more effective, since prefetched data is found in L2 instead of main memory. Consequently, L1 is able to capture more accesses to this region of memory. In fact, we verify that the L1 MPKI receives an overall reduction of 7.7% for all of *art*.

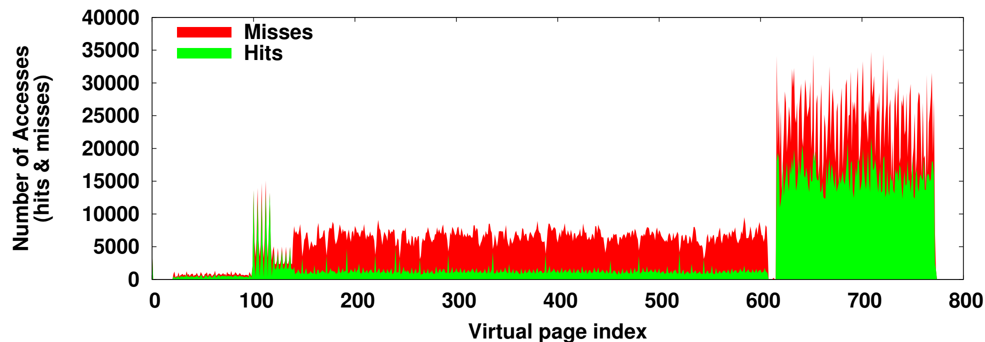
It is also important to observe the impact of restricting the leftmost memory region to the pollute buffer. In this particular case, the memory region did not suffer an increase in L2 miss rate, as may have been expected. In fact, a reduction of 1% was measured, as ROCS kept some pages from the leftmost region in the non-pollute partition of the cache. In essence, the hits seen in the leftmost memory region are primarily due to the short-term reuse of lines from the bus and tds arrays. The pollute buffer, while quite small, is still able to cache these lines for a short period of time, enough to allow reuse of lines with bursty accesses. This fact illustrates a fundamental difference between the pollute buffer and cache bypassing based approaches for addressing cache pollution at the last-level cache [68, 149]. (A discussion on cache bypassing was presented in the Section 2.1.8).

3.6.4 Case study: *swim*

Out of all benchmarks evaluated, *swim* observed the highest performance improvement (34% speedup on execution run-time). Perhaps surprisingly, the performance improvement was not a result of L2 MPKI reduction of *swim*'s data; in fact, a breakdown of stall cycles shows that the performance



a) Default Linux (no cache filtering)



b) ROCS (with cache filtering)

Figure 3.15: Page-level cache miss rate characterization for *art*. The histogram on the top shows the execution with no filtering, and on the bottom, the effects of filtering are shown.

difference comes from handling TLB misses. For *swim*, TLB miss handling contributes to 55% of the stalls when run under Linux. With ROCS, the stalls due to TLB misses decrease to 29% of the total stall cycles.

The main reason for the reduction of TLB miss-handling stalls is that the reduced L2 cache pollution with ROCS allows the hardware page-table walker to find more page-table entries in the L2, incurring fewer full main memory stalls. The PMU in the PowerPC 970FX processor does not contain support for counting the source (in the memory hierarchy) of page-table entries. However, the PMU does provide an event to count the number of cycles used by the hardware page-table walker. Our experiments show that TLB miss handling under ROCS is 69% faster than on Linux without ROCS (an average of 136 cycles per page-table walk on ROCS versus 443 cycles without it).

This illustrates another source of performance benefit from reducing secondary or last-level cache pollution. This case study shows that ROCS also reduces cache pollution effects on metadata in the L2, mitigating the interference with infrequently accessed, but performance critical data, such as page-table entries.

3.7 Discussion

3.7.1 Limitations

In this chapter, we have demonstrated that operating system cache filtering, through the use of a pollute buffer, improves the performance of memory intensive workloads. However, we envision that the technique presented has limited applicability to workloads with specific characteristics. In particular, the fundamental behaviors this work relies on are:

1. Homogeneous page-level access pattern. The cache filtering technique we propose is limited to classifying entire pages and cannot identify, nor filter, specific cache-lines. Although the applications used in this study exhibit homogeneous patterns in regions that are even longer than a single page, we have not attempted to perform a comprehensive analysis of workloads to identify applications that do not show this behavior.

   For applications that collocate low reuse and high reuse data within a single page, page level cache filtering will likely be ineffective, and potentially harmful to performance. A potential solution to this problem is to influence the placement of data items within the *virtual* address space, so that items with similar reuse patterns are collocated within the virtual address space. For some applications, statically modifying the memory allocator may be sufficient, allowing allocation to be specialized per data type, for example.

2. Regions of address space that display differing reuse. Filtering pages or regions of the application address space is expected to be useful only if there are other regions that exhibit shorter reuse distances. In this case, reducing the cache space that is allocated to low reuse regions is beneficial as there are other regions that can better use the cache. For applications that have a uniform access pattern throughout their address space, we expect that filtering will yield modest or negligible benefits.

3. Stable access patterns for extended periods of time. As discussed in Section 3.5.1, the overhead for continuous profiling using hardware performance counters is significant. We overcome the high overhead of profiling by only sampling the address space when the application starts and when a phase change is detected. For applications that change execution behavior frequently, filtering will not be able to recover the overhead of profiling, and our technique may degrade overall performance.

   In the future, we hope that processors will include improved hardware performance counter functionality that allows for monitoring of address spaces at run-time with significantly lower overhead. One example of a feature that could reduce the overhead of address space profiling is a feature recently included in Intel x86 processors, called "debug store mechanism" [90]. This mechanism allows all of the architecturally visible registers to be spilled to a reserved

area of memory without issuing an interrupt to the software. While the debug store mechanism saves a lot more state than necessary for address space profiling, it incurs less overhead to execution than requiring an interrupt for every sample of data collected.

3.7.2 Stall-rate oriented profiling

In the work presented in this chapter, a principal metric used to determine per-page cache effectiveness was *miss rate*. Miss rate was used to distinguish the cache utility of the pages pertaining to an application, allowing our system to segregate pages into pollute and non-pollute pages.

A potentially more direct metric to measure the performance impact of accessing each page is the number of stalls observed at the pipeline level due to long latency memory accesses. Using the measured number of stalls, instead of number of misses, can more precisely measure the impact each miss has on application performance. Several features of modern processors can make the impact of different cache misses to be distinct. For example, the out-of-order engine is able to overlap useful computation with part, or all, of the latency of retrieving a cache item from a higher level cache, causing variation on the number of stalls observed by the pipeline. Furthermore, items that have been prefetched in advance, but not yet placed in a cache level, are considered to unavailable (a miss), but are retrieved faster than other unavailable items. Equating these different classes of misses can lead to an incorrect ordering of pages with respect to their effective use of cache space.

In previous work, we have successfully used a *stall rate* metric to predict the impact of partitioning a shared cache on a multi-core processor [186]. We believe it should be possible to apply stall rates to more precisely determine the performance impact of accessing each application page. Unfortunately, the performance monitoring unit of the processor we used in this work (PowerPC 970FX) did not provide this type of feedback. However, preliminary experiments on a subsequent version of the PowerPC processor (Power5) indicates that this could be a potential improvement over the model described solely relying on miss rates.

3.7.3 Software managed/assisted processor caches

The memory performance gap affects the performance of several classes of applications, and as application footprints grow, this impact is unlikely to disappear. As the shrinkage of transistor feature sizes is expected to continue, hardware vendors are growing the size of processor caches with each generation. In the next few years, mainstream processors are expected to incorporate caches of more than 32 megabytes in size. As a historical reference, this was the typical size of mainstream computer main memory around 1995, which was managed by the operating system.

We argue that there are advantages in allowing the operating system to cooperate in managing processor caches, particularly the larger and slower levels of the cache hierarchy. Along with the ROCS technique described in this chapter, there are other potential advantages in allowing runtime software to assist in managing the processor cache hierarchy:

1. Reducing HW complexity. A major concern in developing new processors is the time and cost required to design and verify new features. In the context of caching, despite the known weaknesses of LRU based algorithms, modern processors still employ basic LRU policies, such as the bit-PLRU algorithm that requires a single bit per cache line of metadata.

   Having software assist in managing caches allows for more complex caching policies, such as the one we developed, without requiring significant overheads in designing new processor caches. The primary cost would then be the design and implementation of an initial interface that provides software with better mechanisms for both monitoring and controlling cache replacement policies.

2. Software-level semantic information. One crucial difference between implementing caching policies in hardware and implementing caching policies in the run-time software stack is availability of semantic information. While the operating system inherently tracks processes, address spaces, and memory allocations, processors are generally oblivious to these software-level constructs. For this reason, the operating system is uniquely positioned in the compute stack to assist caching.

   An example of this advantage is illustrated by the technique described in this chapter. Coarse-grain cache behavior of address-spaces is reasonable to track in the operating system, since the operating system already has structures to track other address space characteristics. But other information available at the software level may prove useful for caching policies, such as program and run-time system memory layout, thread scheduling, external inputs (I/O), etc.

3.8 Summary

The *memory wall* problem has been studied intensively in the computer architecture and software communities and has resulted in a wide range of proposals for reducing the effects of memory latency on performance. Chip makers, for example, are dedicating an increasing number of transistors for larger on and off-chip caches. However, not all workloads have responded to this increase with corresponding performance or hit ratio improvements.

We argued that proper management of the memory hierarchy is becoming more critical to achieve good performance and that software can play a significant and fruitful role in managing this hierarchy. We believe there are new opportunities to be explored with tighter cooperation between run-time software systems and the underlying hardware. This work presents a concrete example of this type of cooperation.

In this chapter, we focused on attacking the specific problem of cache pollution in secondary-level caches for applications that exhibit *intra-application interference*. We observed that cache behavior, and pollution in particular, is uniform within a memory region, typically spanning multi-

ple memory pages of application address space. We described the use of hardware performance counters, present on current hardware, to classify memory pages with respect to pollution.

We introduced the concept of a *pollute buffer* to host cache lines of pages with little or no reuse before eviction. We demonstrated how the secondary-level cache can be partitioned with operating system page coloring to provide a pollute buffer within the cache. This technique requires no additional hardware support and no modifications to application code or binary.

Using these concepts, we described a complete implementation of a run-time operating system cache-filtering service (ROCS). We evaluated the performance of our system on seven memory intensive SPEC CPU 2000 and NAS benchmarks, showing performance improvements of up to 34% on run-time execution, with 16% on average.

The system we implemented makes extensive use of processor performance monitoring units (PMU). Unfortunately, the architecture and interfaces of PMUs are substantially different for each processor family and in fact different across different processors within the same family. Standardizing the key PMU components and interfaces would, in our opinion, greatly accelerate the development and ubiquity of additional software optimizations, similar to the one we described in this chapter. The impact of the IEEE 754 floating-point standardization efforts of 30 years ago should provide good motivation.

In conclusion, this work explored the use of rudimentary processor interfaces for monitoring and managing secondary-level caches. We believe that with better mechanisms for cooperation between hardware and software layers, further opportunities for improving performance would arise. We hope that this work serves as encouragement to hardware designers to include and expose more flexibility in processor components to the software layer.

Chapter 4

Exception-less System Calls

For the past 30+ years, system calls have been the *de facto* interface used by applications to request services from the operating system kernel. System calls have almost universally been implemented as a *synchronous* mechanism, where a special processor instruction is used to yield user-space execution to the kernel. In the first part of this chapter, we evaluate the performance impact of traditional synchronous system calls on system intensive workloads. We show that synchronous system calls negatively affect performance in a significant way, primarily because of pipeline flushing and pollution of key processor structures (e.g., TLB, data and instruction caches, etc.).

In this chapter, we propose a new mechanism for applications to request services from the operating system kernel: *exception-less system calls*. They improve processor efficiency by enabling flexibility in the scheduling of operating system work, which in turn can lead to significantly increased temporal and spacial locality of execution in both user and kernel space, thus reducing pollution effects on processor structures. Exception-less system calls are particularly effective on multicore processors.

In the subsequent two chapters, we explore different ways for applications to make use of the new exception-less system call mechanism. The first way, which is completely transparent to the application, uses multi-threading to hide asynchronous communication between the operating system kernel and the application. In the second way, we show how applications can directly use the exception-less system call interface by designing programs that follow an event-driven architecture.

4.1 Introduction

System calls are the *de facto* interface to the operating system kernel. They are used to request services offered by, and implemented in the operating system kernel. While different operating systems offer a variety of different services, the basic underlying system call mechanism has been common on all commercial multiprocessed operating systems for decades. System call invocation typically involves writing arguments to appropriate registers and then issuing a special machine

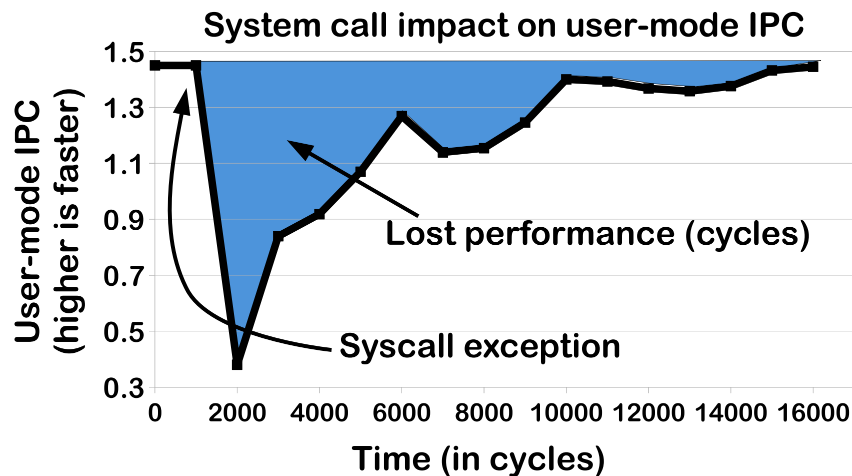


Figure 4.1: User-mode instructions per cycles (IPC) of Xalan (from SPEC CPU 2006) in response to a system call exception event, as measured on an Intel Core i7 processor.

instruction that raises a synchronous exception, immediately yielding user-mode execution to a kernel-mode exception handler. Two important properties of the traditional system call design are: (1) a processor exception (also known as software interrupt) is used to communicate with the kernel, and (2) a synchronous execution model is enforced, as the application expects the completion of the system call before resuming user-mode execution. Both of these properties result in performance inefficiencies on modern processors.

The increasing number of available transistors on a chip (Moore's Law) has, over the years, led to increasingly sophisticated processor structures, such as superscalar and out-of-order execution units, multi-level caches, and branch predictors. These processor structures have, in turn, led to a large increase in the performance *potential* of software, but at the same time there is a widening gap between the performance of efficient software and the performance of inefficient software, primarily due to the increasing performance disparity of different structures used to store and access data (e.g., registers vs. caches vs. memory). Server and system-intensive workloads, which are of particular interest in our work, are known to perform well below the potential processor throughput [112, 114, 160]. Most studies attribute this inefficiency to the lack of locality. *We claim that part of this lack of locality, and resulting performance degradation, stems from the current synchronous system call interface.*

Synchronous implementation of system calls negatively impacts the performance of system intensive workloads, both in terms of the *direct* costs of mode switching and, more interestingly, in terms of the *indirect* pollution of important processor structures which affects both user-mode and kernel-mode performance. A motivating example that quantifies the impact of system call pollution on application performance can be seen in Figure 4.1. It depicts the user-mode instructions per cycles (kernel cycles and instructions are ignored) of one of the SPEC CPU 2006 benchmarks (Xalan) immediately before and after a `pwrite` system call. There is a significant drop in instructions per cycle (IPC) due to the system call, and it takes up to 14,000 cycles of execution before the

IPC of this application returns to its previous level. As we will show, this performance degradation is mainly due to interference caused by the kernel on key processor structures.

To improve locality in the execution of system intensive workloads, we propose a new operating system mechanism: the **exception-less system call**. An exception-less system call is a new mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. In our implementation, system calls are issued by writing kernel requests to a reserved **syscall page** shared between the application and the kernel, using normal memory store operations. The actual execution of system calls is performed asynchronously by special in-kernel **syscall threads**, which post the results of system calls to the syscall page after their completion.

Decoupling the system call execution from its invocation makes flexible system call scheduling possible, offering optimizations along two dimensions. The first optimization allows for the deferred batch execution of system calls resulting in increased temporal locality of execution. The second provides the ability to execute system calls on a separate core, in parallel to executing user-mode threads, resulting in spatial, per core locality. In both cases, system call threads become a simple, but powerful abstraction.

One interesting feature of the proposed decoupled system call model is the possibility of *dynamic* core specialization in multicore systems. Cores can become temporarily specialized for either user-mode or kernel-mode execution, depending on the current system load. We describe how the operating system kernel can dynamically adapt core specialization to the demands of the workload.

One important challenge of our proposed system is how to best use the exception-less system call interface. In the following two chapters, we explore different strategies to help programmers make use of exception-less system calls so that overall performance is improved.

4.2 The (Real) Costs of System Calls

In this section, we analyze the performance costs associated with a traditional, synchronous system call. We analyze these costs in terms of mode switch time, the system call footprint, the effect on user-mode IPC, and the effect on kernel-mode IPC. We used the Linux operating system kernel and an Intel Nehalem (Core i7) processor, along with its performance counters to obtain our measurements. However, we believe the lessons learned are applicable to most modern high-performance processors[1] and other operating system kernels.

4.2.1 Mode Switch Cost

Traditionally, the performance cost attributed to system calls is the *mode switch time*. The mode switch time consists of the time required to execute the appropriate system call instruction in user-

[1] Experiments performed on an older PowerPC 970 processor yielded similar insights than the ones presented here.

Processor	Pentium 4 2.8 GHz	PPC 970 2.3 GHz	Power5 1.5 GHz	Core2 1.6 GHz	Core i7 2.2 GHz
Year	2000	2002	2004	2006	2008
Null syscall time (cycles)	447	344	387	237	150

Table 4.1: Micro-benchmark results of null system call overhead for a variety of processors. Columns are ordered by release date of the processor.

mode, resuming execution in an elevated protection domain (kernel-mode), and the return of control back to user-mode. Modern processors implement the mode switch as a processor exception: flushing the user-mode pipeline, saving a few registers onto the kernel stack, changing the protection domain, and redirecting execution to the registered exception handler. Subsequently, return from exception is necessary to resume execution in user-mode. At the processor level, the return also results in a mode switch that uses a similar process as the system call exception (pipeline flush, change in protection domain, fetch of new instructions, etc.).

Over the years, both operating system developers and hardware designers have focused on reducing the cost of performing mode switches. A sample of the cost of entering and subsequently exiting kernel mode, in a tight loop so that instructions and data are fetched from the L1 cache, is listed in Table 4.1. For example, the mechanism used for mode switching in x86 based processors have had two versions in the recent past. In one version, a generic interrupt instruction was used to notify the operating system of a system call request. This mechanism, because it is generic, required several memory accesses to verify parameters previously configured in an interrupt descriptor table, and resulted in several hundred cycle latency (Pentium 4 in Table 4.1). Intel later introduced an optimized instruction specifically for implementing system calls, called *sysenter/sysexit*. Although it allows for less customization, it resulted in a saving of over 200 cycles, as the results for the Core2 show.

To measure more detailed mode switching costs on our main experimental platform, we measured the mode switch time by implementing a new system call, `gettsc`. It obtains the timestamp counter of the processor and immediately returns to user-mode. We created a simple benchmark that invoked `gettsc` 1 billion times in a tight loop, recording the timestamp before and after each call. The difference between each of the three timestamps — the timestamp immediately before the system call, the timestamp obtained by the system call, and the timestamp immediately after the system call — identifies the number of cycles necessary to enter and leave the operating system kernel, namely 79 cycles and 71 cycles, respectively. The total round-trip time for the `gettsc` system call is modest at 150 cycles, being less than the latency of a memory access that misses the processor caches (250 cycles on our machine).[2]

[2]For all experiments presented in this chapter, user-mode applications execute in 64-bit mode and when using synchronous system calls, use the "syscall" x86_64 instruction, which is currently the default in Linux.

Syscall	Instructions	Cycles	IPC	i-cache	d-cache	L2	L3	d-TLB
`stat`	4972	13585	0.37	32	186	660	2559	21
`pread`	3739	12300	0.30	32	294	679	2160	20
`pwrite`	5689	31285	0.18	50	373	985	3160	44
`open+close`	6631	19162	0.34	47	240	900	3534	28
`mmap+munmap`	8977	19079	0.47	41	233	869	3913	7
`open+write+close`	9921	32815	0.30	78	481	1462	5105	49

Table 4.2: System call footprint of different processor structures. For the processors structures (caches and TLB), the numbers represent number of entries evicted; the cache line for the processor is of 64-bytes. i-cache and d-cache refer to the instruction and data sections of the L1 cache, respectively. The d-TLB represents the data portion of the TLB.

4.2.2 System Call Footprint

The mode switch time, however, is only part of the cost of a system call. During kernel-mode execution, processor structures including the L1 data and instruction caches, translation look-aside buffers (TLB), branch prediction tables, prefetch buffers, as well as larger unified caches (L2 and L3), are populated with kernel specific state. The replacement of user-mode processor state by kernel-mode processor state is referred to as the processor state *pollution* caused by a system call.

To quantify the pollution caused by system calls, we used the Core i7 hardware performance counters (HPC). We ran a high instruction per cycle (IPC) workload, Xalan, from the SPEC CPU 2006 benchmark suite that is known to invoke few system calls. We configured an HPC to trigger infrequently (once every 10 million user-mode instructions) so that the processor structures would be dominated with application state. We then set up the HPC exception handler to execute specific system calls, while measuring the replacement of application state in the processor structures caused by kernel execution (but not by the performance counter exception handler itself). To isolate the replacements caused by system call execution from the replacements caused by the HPC interrupt handler, we executed the same experiments with two configurations. In the first configuration, we enabled HPC interrupts while not executing other system work. In the second configuration, in addition to the interrupt handler we also executed system work. We report the difference between the two executions which corresponds to the replacements caused by the system call alone.

Table 4.2 shows the footprint on several processor structures for three different system calls and three system call combinations. The data shows that, even though the number of i-cache lines replaced is modest (between 2 and 5 KB), the number of d-cache lines replaced is significant. Given that the size of the d-cache on this processor is 32 KB, we see that the system calls listed pollute at least half of the d-cache, and almost all of the d-cache in the "open+write+close" case. The 64 entry first level d-TLB is also significantly polluted by most system calls. Finally, it is interesting to note that the system call impact on the L2 and L3 caches is larger than on the L1 caches, primarily because the L2 and L3 caches use more aggressive prefetching.

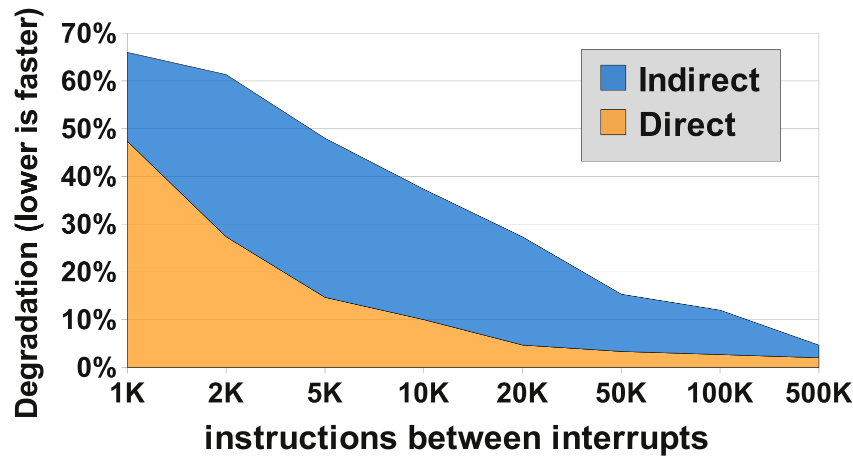


Figure 4.2: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for Xalan. The bottom portion of the graph shows the impact on user-mode IPC by simply raising interrupts, which are minimally serviced by the operating system (labeled as *direct*). The top portion of the graph shows the impact on user-mode IPC by also executing a `pwrite` system call within the interrupt handler.

4.2.3 System Call Impact on User IPC

Ultimately, the most important measure of the real cost of system calls is the performance impact on the application. To quantify this, we executed an experiment similar to the one described in the previous subsection. However, instead of measuring kernel-mode events, we only measured user-mode instructions per cycle (IPC), ignoring all kernel execution. Ideally, user-mode IPC should not decrease as a result of invoking system calls, since the cycles and instructions executed as part of the system call are ignored in our measurements. In practice, however, user-mode IPC is affected by two sources of overhead:

Direct: The processor exception associated with the system call instruction that flushes the processor pipeline.

Indirect: System call pollution on the processor structures, as quantified in Table 4.2.

Figures 4.2 and 4.3 show the degradation in user-mode IPC when running Xalan (from SPEC CPU 2006) and SPEC-JBB 2005, respectively, given different frequencies of `pwrite` calls. These benchmarks were chosen since they have been created to avoid significant use of system services, and should spend only 1-2% of time executing in kernel-mode. The graphs show that different workloads can have different sensitivities to system call pollution. Xalan has a baseline user-mode IPC of 1.46, but the IPC degrades by up to 65% when executing a `pwrite` every 1,000-2,000 instructions, yielding an IPC between 0.58 and 0.50. SPEC-JBB has a slightly lower baseline of 0.97, but still degrades user-mode IPC by 45%.

The figures also depict the breakdown of user-mode IPC degradation due to direct and indirect costs. The degradation due to the direct cost was measured by issuing a null system call, while the

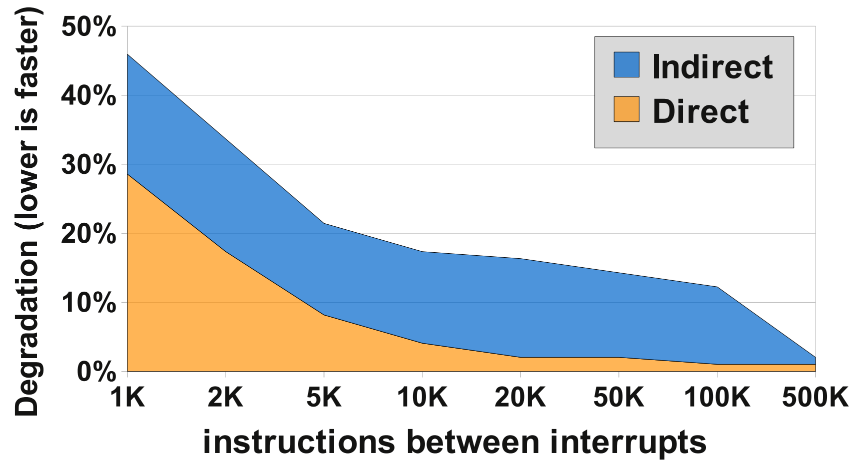


Figure 4.3: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for SPEC JBB 2005. The bottom portion of the graph shows the impact on user-mode IPC by simply raising interrupts, which are minimally serviced by the operating system (labeled as *direct*). The top portion of the graph shows the impact on user-mode IPC by also executing a `pwrite` system call within the interrupt handler.

Workload (server)	**Application instructions between syscalls**
DNSbench (BIND)	2445
ApacheBench (Apache)	3368
Sysbench (MySQL)	12435

Table 4.3: The average number of instructions executed on different workloads before issuing a syscall.

indirect portion is calculated subtracting the direct cost from the degradation measured when issuing a `pwrite` system call. For high frequency system call invocation (once every 2,000 instructions, or less), the direct cost of raising an exception and subsequent flushing of the processor pipeline is the largest source of user-mode IPC degradation. However, for medium frequencies of system call invocation (once per 2,000 to 100,000 instructions), the *indirect* cost of system calls is the dominant source of user-mode IPC degradation.

To understand the implication of these results on typical server workloads, it is necessary to quantify the system call frequency of these workloads. The average user-mode instruction count between consecutive system calls for three popular server workloads are shown in Table 4.3. For this frequency range in Figures 4.2 and 4.3 we observe user-mode IPC performance degradation between 20% and 60%. While the execution of the server workloads listed in Table 4.3 is not identical to that of Xalan or SPEC-JBB, the data presented here indicates that server workloads suffer from significant performance degradation due to processor pollution of system calls.

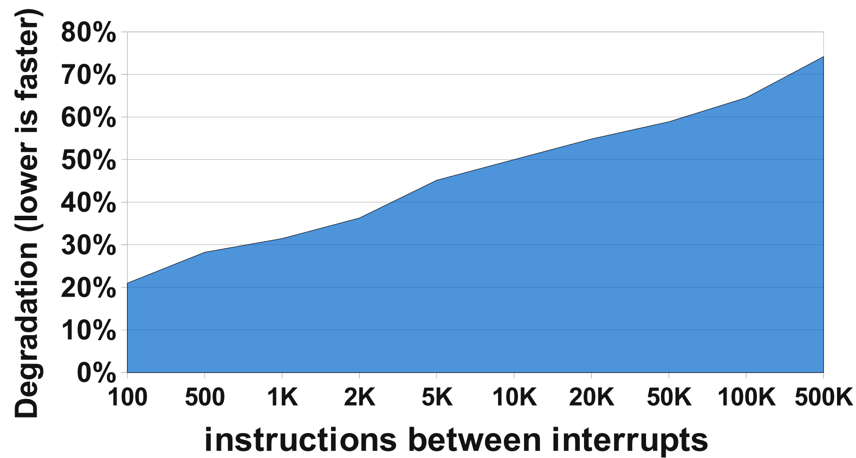


Figure 4.4: System call (`pwrite`), impact on kernel-mode IPCs for x as a function of system call frequency. The graph shows the impact on kernel-mode IPC of raising an interrupt and executing a `pwrite` system call within the interrupt handler.

4.2.4 Mode Switching Cost on Kernel IPC

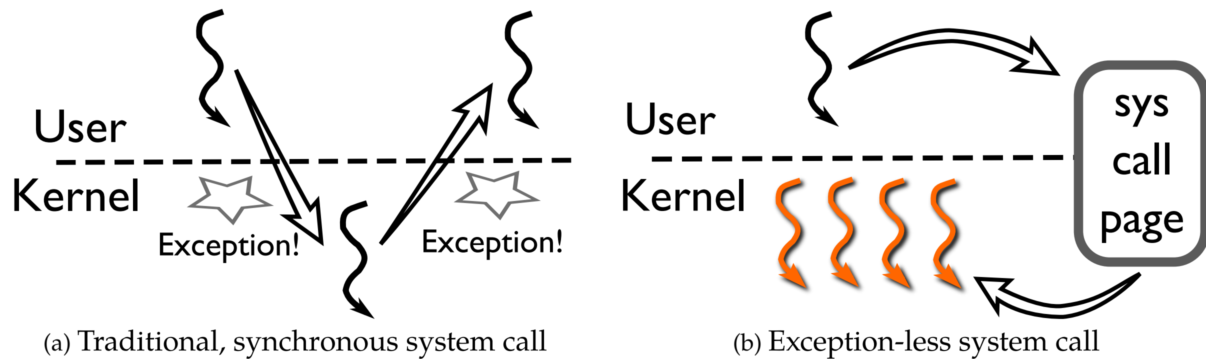
The lack of locality due to frequent mode switches also negatively affects kernel-mode IPC. Figure 4.4 shows the impact of different system call frequencies on the kernel-mode IPC. As expected, the performance trend is opposite to that of user-mode execution. The more frequent the system calls, the more kernel state is maintained in the processor.

Note that the kernel-mode IPC listed in Table 4.2 for different system calls ranges from 0.18 to 0.47, with an average of 0.32. This is significantly lower than the 1.47 and 0.97 user-mode IPC for Xalan and SPEC-JBB 2005, respectively; up to *8x* slower.

4.2.5 Significance of system call interference experiments

The experiments shown in this section do not demonstrate an upper or lower bound on the performance impact of multiplexing application and operating system execution at fine granularity. The impact of operating system interference depends on several factors, some of which are specific to the processor architecture and some are application specific. One example of this variation is shown in Figures 4.2 and 4.3, where the same experiment yielded different performance degradations depending on the application. In particular, we expect that the applications we target in this work, server class workloads with frequent use of operating system services, will exhibit access patterns that are different than those of Xalan or Spec JBB 2005.

Nonetheless, these experiments do show that for the range of system call requests listed in Table 4.3 it is expected that the interference is a significant source of inefficiency. Furthermore, these experiments are the first, to the best of our knowledge, to attempt to quantify the different sources of operating system overhead, and to provide detailed performance counter information, on a real processor, on the effect of a system call on various performance sensitive proccessor structures.



(a) Traditional, synchronous system call (b) Exception-less system call

Figure 4.5: Illustration of synchronous (a) and exception-less (b) system call invocation. The wavy lines represent threads of execution (user or kernel). The left diagram illustrates the sequential nature of exception-based system calls. When an application thread makes a system call, it uses a special instruction that generates a processor interrupt or exception. The processor immediately transfers control to the operating system kernel, where the call is executed synchronously. After which, the kernel returns control to the application thread, which is done through an exception-based mechanism similar to the system call. The right diagram, on the other hand, depicts exception-less user and kernel communication. Messages are exchanged asynchronously through portion of shared memory, which we call *syscall page*, by simply reading from and writing to it.

Finally, the insights obtained from these experiments were crucial in guiding the design of an alternative, more efficient, system call mechanism, namely, the exception-less system call mechanism which we present in the following section.

4.3 Exception-Less System Calls

To address (and partially eliminate) the performance impact of traditional, synchronous system calls on system intensive workloads, we propose a new operating system mechanism called **exception-less system call**. Exception-less system call is an asynchronous mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. Instead of using a synchronous notification mechanism where registers are used to communicate argument values to the operating system kernel using a processor interrupt or exception, exception-less system calls uses shared memory to communicate with the kernel, with no synchronous notification. Figure 4.5 depicts the contrast between a traditional synchronous system call and the proposed exception-less system call mechanism.

The key benefit of exception-less system calls is the flexibility in scheduling system call execution, ultimately providing improved locality of execution for both user and kernel code. Here we explore two use cases:

- **System call batching:** Delaying the execution of a series of system calls and executing them in batches minimizes the frequency of switching between user and kernel execution, eliminating some of the mode switch overhead and allowing for improved *temporal locality*. This

improves both the direct and indirect costs of system calls.

- **Core specialization:** In multicore systems, exception-less system calls allow a system call to be scheduled on a core different than the one on which the system call was invoked. Scheduling system calls on a separate processor core allows for improved *spatial locality* and with it lower indirect costs. In an ideal scenario, no mode switches are necessary, eliminating the direct cost of system calls.

The design of exception-less system calls consists of two components: (1) an exception-less interface for user-space threads to register system calls, along with (2) an in-kernel threading system that allows the delayed (asynchronous) execution of system calls, without interrupting or blocking the thread in user-space.

4.3.1 Exception-Less Syscall Interface

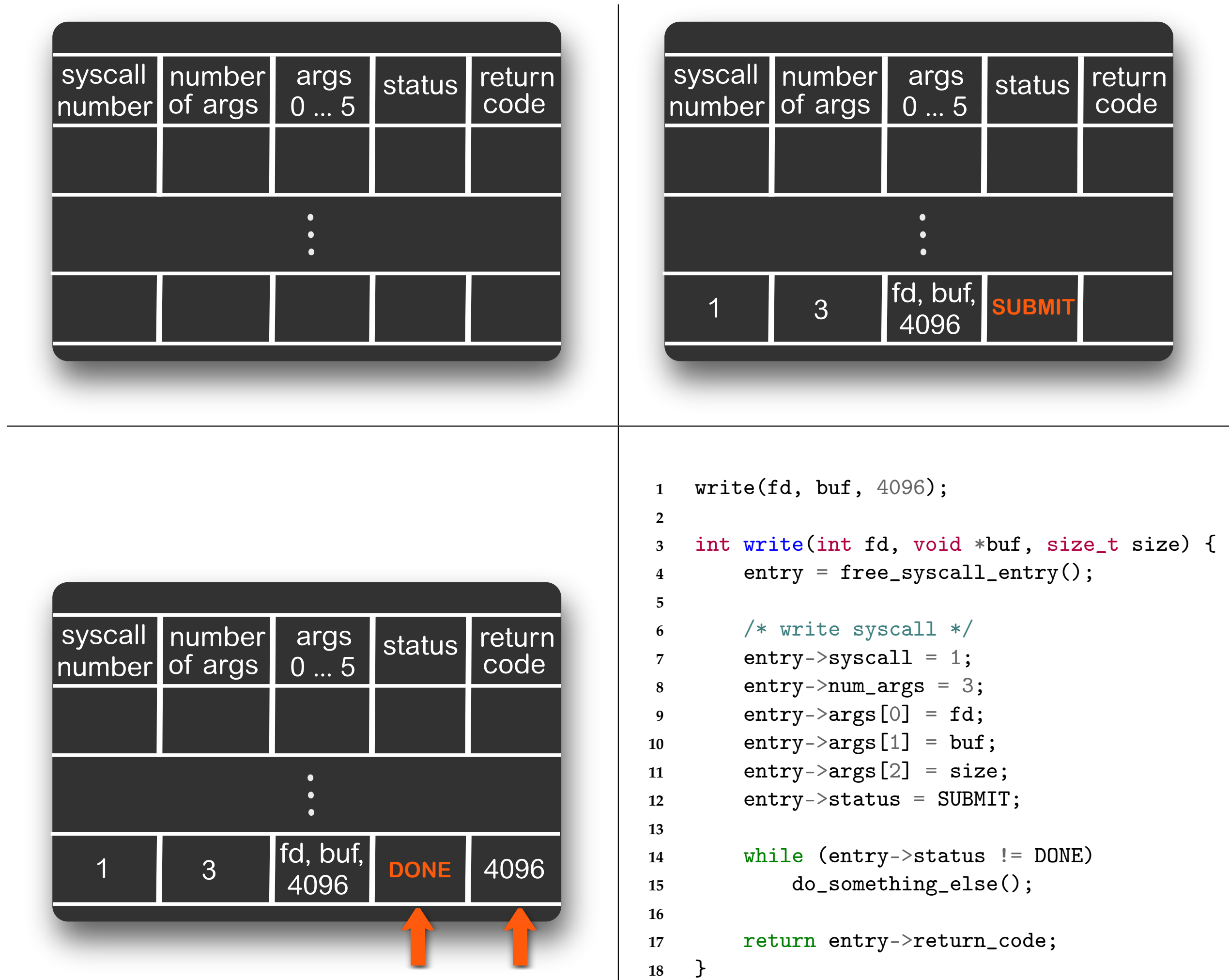
The interface for exception-less system calls is simply a set of memory pages that is shared between user and kernel space. The shared memory page, henceforth referred to as **syscall page**, is organized to contain exception-less system call entries. Each entry contains space for the request status, system call number, arguments, and return value.

With traditional synchronous system calls, invocation occurs by populating predefined registers with system call information and issuing a specific machine instruction that immediately raises an exception. In contrast, to issue an exception-less system call, the user-space threads must find a free entry in the syscall page and populate the entry with appropriate values using regular store instructions. The user-space thread can then continue executing without interruption. It is the responsibility of the user-space thread to later verify the completion of the system call by reading the status information in the entry. None of these operations, issuing a system call or verifying its completion, causes exceptions to be raised.

4.3.2 Syscall Pages

Syscall pages can be viewed as a table of syscall entries, each containing information specific to a single system call request, including the system call number, arguments, status (free, submitted, busy, or done), and the result. To top-left diagram of Figure 4.6 depicts the organization of the syscall page. In our 64-bit implementation, we organized each entry to occupy 64 bytes. This size comes from the Linux ABI which allows any system call to have up to 6 arguments, and a return value, totaling 56 bytes. Although the remaining 3 fields (syscall number, status and number of arguments) could be packed in less than the remaining 8 bytes, we selected 64 bytes because 64 is a divisor of popular cache line sizes of today's processor.

To issue an exception-less system call, the user-space thread must find an entry in one of its syscall pages that contain a *free* status field. It then writes the syscall number and arguments to




```
1  write(fd, buf, 4096);
2
3  int write(int fd, void *buf, size_t size) {
4      entry = free_syscall_entry();
5
6      /* write syscall */
7      entry->syscall = 1;
8      entry->num_args = 3;
9      entry->args[0] = fd;
10     entry->args[1] = buf;
11     entry->args[2] = size;
12     entry->status = SUBMIT;
13
14     while (entry->status != DONE)
15         do_something_else();
16
17     return entry->return_code;
18 }
```

Figure 4.6: Illustration of syscall page, along with expected steps when making an exception-less system call. The top-left figure shows the organization of a syscall page, composed of 64 syscall entries. Each entry contains arguments of the system call (syscall number, number of arguments, and 6 arguments), a status field used to manage the entries and the return code. The bottom-right shows code that could be used to issue an exception-less *write()* system call. The execution of lines 4 through 12 are reflected in the syscall page of the top-right diagram. After the execution of the system call, the operating system kernel updates the status and return code fields of the syscall entry, which can be seen in the bottom-left diagram. At this point, the application will exit the loop (line 14), and can read the return value and exit (line 17).

the entry. Lastly, the status field is changed to *submitted*, indicating to the kernel that the request is ready for execution. User-space must update the status field last, with an appropriate memory barrier, to prevent the kernel from selecting incomplete syscall entries to execute. The thread must then check the status of the entry until it becomes *done*, consume the return value, and finally set the status of the entry to *free*. An example of simple code to make an exception-less system call (a *write()* call), along with the modifications to the syscall page can be visualized in Figure 4.6 (bottom-right).

In Section 4.4.3 we further discuss allocation strategies for syscall pages as well as procedures to cope with the lack of free syscall entries.

4.3.3 Decoupling Execution from Invocation

Along with the exception-less interface, the operating system kernel must support delayed execution of system calls. Unlike exception-based system calls, the exception-less system call interface does not result in an explicit kernel notification, nor does it provide an execution stack. To support decoupled system call execution, we use a special type of kernel thread, which we call **syscall thread**. Syscall threads always execute in kernel mode, and their sole purpose is to pull requests from syscall pages and execute them on behalf of the user-space thread.

The combination of the exception-less system call interface and independent syscall threads allows for great flexibility in the scheduling the execution of system calls. Syscall threads may wake up only after user-space is unable to make further progress, in order to achieve temporal locality of execution on the processor. Orthogonally, syscall threads can be scheduled on a different processor core than that of the user-space thread, allowing for spatial locality of execution. On modern multicore processors, cache to cache communication is relatively fast (in the order of 10s of cycles), so communicating the entries of syscall pages from a user-space core to a kernel core, or vice-versa, should only cause a small number of processor stalls.

4.4 Implementation – FlexSC

Our implementation of the exception-less system call mechanism is called **FlexSC** (Flexible System Call scheduling) and was prototyped as an extension to the Linux kernel. Although our implementation was influenced by a monolithic kernel architecture, we believe that most of our design could be effective with other kernel architectures, e.g., exception-less micro-kernel IPCs, and hypercalls in a paravirtualized environment.

We have implemented FlexSC for the x86_64 and PowerPC64 processor architectures. Porting FlexSC to other architectures is trivial; a single function is needed, which moves arguments from the syscall page to appropriate registers, according to the ABI of the processor architecture.

Two new system calls were added to Linux as part of FlexSC, `flexsc_register` and `flexsc_wait`.

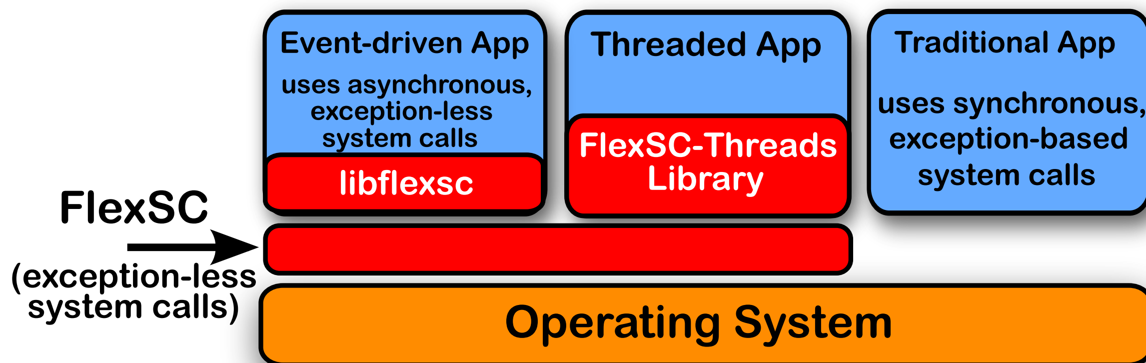


Figure 4.7: Component-level overview of FlexSC. The implementation of operating system services, representative by the bottom box, are not altered by our FlexSC system. As a consequence, legacy applications that use exception-based system call mechanism continue to work unaltered. We introduce a new operating system mechanism, exception-less system calls (FlexSC), that can be used by applications to asynchronously request operating system services. We also introduce two new libraries, FlexSC-Threads which is intended to support legacy thread based programs in a transparent way, and *libflexsc* which supports event-driven applications that directly make use of FlexSC.

4.4.1 `flexsc_register()`

This system call is used by processes that wish to use the FlexSC facility. Making this registration procedure explicit is not strictly necessary, as processes can be registered with FlexSC upon creation. We chose to make it explicit mainly for convenience of prototyping, giving us more control and flexibility in user-space. One legitimate reason for making registration explicit is to avoid the extra initialization overheads incurred for processes that do not use exception-less system calls.

Invocation of the `flexsc_register` system call must use the traditional, exception-based system call interface to avoid complex bootstrapping; however, since this system call needs to execute only once, it does not impact application performance. Registration involves two steps: mapping one or more syscall pages into user-space virtual memory space, and spawning a number of syscall threads (discussed in more detail in Section 4.4.4).

4.4.2 `flexsc_wait()`

The decoupled execution model of exception-less system calls creates a challenge in user-space execution, namely what to do when the user-space thread has nothing more to execute and is waiting on pending system calls. With the proposed execution model, FlexSC loses the ability to determine when a user-space thread should be put to sleep. With synchronous system calls, this is simply achieved by putting the thread to sleep while it is executing a system call if the call blocks waiting for a resource.

A naive solution is to allow the user-space thread to busy wait, checking the status entries in the syscall page for a system call to be marked as "*done*". Busy waiting, despite its simplicity, is

unacceptably inefficient. A second strategy is to block the user-mode thread when all of the syscall threads executing system calls on its behalf are sleeping or blocked, and no more entries are registered in the syscall page. This option would avoid inefficiencies of busy waiting, but could potentially yield a user-mode thread that is performing useful work; simply because all of its system calls are blocked, does not imply that the user thread is not currently executing useful computations.

The solution we adopted is to require that the user explicitly communicate to the kernel that it cannot progress until one of the issued system calls completes by invoking the `flexsc_wait` system call. We implemented `flexsc_wait` as an exception-based system call, since execution should be synchronously directed to the kernel. FlexSC will later wake up the user-space thread when at least one of posted system calls are complete.

4.4.3 Syscall Page Allocation

Syscall pages are allocated and mapped to both kernel and application address spaces at registration time. As described in Section 4.3.2, each syscall page is divided into 64 entries. However, certain applications execute more than 64 concurrent system calls and therefore mapping a single syscall page per process would pose as a limitation to these applications. For this reason, we allow applications to request several syscall pages during registration with FlexSC. Specifically for applications with a large number of threads (described in Chapter 5), allowing applications to allocate several syscall pages results in improved performance.

In some cases, it may be difficult for applications to foretell the number of concurrent system calls it will make at run-time. Underestimating the number of concurrent system calls needed may affect performance since the application must partially serialize system call invocation for the cases when there are no free syscall entries. A more severe problem of underestimating the number of concurrent system calls is a potential for deadlock. If the progress of an application depends on the successful completion of a certain number of concurrent system calls (e.g., a `read()` that blocks until a `write()` is performed), provisioning insufficient entries can prohibit the forward progress of the application.

For these reasons, it may be beneficial to allow applications to change the number of syscall pages at run-time. In theory, this feature should be simple to implement by adding new (synchronous) system calls such as `flexsc_alloc()` and `flexsc_free()`. The logic for allocating and mapping new pages would be akin to the existing logic within `flexsc_register`. Although we have not implemented this feature in our current prototype, there should be no impediments to implement these calls.

In our implementation, to avoid any possibility of deadlock, we have provisioned sufficient system call entries to allow the forward progress of the application in worst case scenarios (e.g., every application thread is concurrently blocked or executing a system call).

4.4.4 Syscall Threads

Syscall threads is the mechanism used by FlexSC to allow for exception-less execution of system calls. The Linux system call execution model has influenced some implementation aspects of syscall threads in FlexSC: (1) the virtual address space in which system call execution occurs is the address space of the corresponding process, and (2) the current thread context can be used to block execution should a necessary resource not be available (for example, waiting for I/O).

To resolve the virtual address space requirement, syscall threads are created during `flexsc_register`. Syscall threads are thus "cloned" from the registering process, resulting in threads that share the original virtual address space. This allows the transfer of data from/to user-space with no modification to Linux's code.

FlexSC would ideally never allow a syscall thread to sleep. If a resource is not currently available, notification of the resource becoming available should be arranged, and execution of the next pending system call should begin. However, implementing this behavior in Linux would require significant changes and a departure from the basic Linux architecture. The current Linux kernel is based on a blocking thread model for contended resources or I/O operations, which is a model also shared by several monolithic operating system kernels. The use of the thread context, including its stack, is a convenient mechanism to allow for operations to be preempted and continued at a later time. Instead of relying on the ability to block thread contexts, Linux could be modified to adopt a fully event-driven architecture where operations are queued for future completion and the kernel run-time provides continuations that advance execution when resources become available [162].

Instead of modifying Linux to support completely asynchronous operations, we adopted a strategy that allows FlexSC to maintain the Linux thread blocking architecture, as well as requiring only minor modifications (3 lines of code) to Linux context switching code, by creating multiple syscall threads for each process that registers with FlexSC. In fact, FlexSC spawns as many syscall threads as there are entries available in the syscall pages mapped in the process. This provisions for the worst case where every pending system call blocks during execution.

Spawning hundreds or thousands of syscall threads may seem expensive, but Linux in-kernel threads are typically much lighter weight than user threads: all that is needed is a `task_struct` and a small, 2-page, stack for execution. All the other structures (page table, file table, etc.) are shared with the user process. In total, only 10KB of memory are needed per syscall thread. We expect this memory overhead to be modest when compared to other memory allocations of the application. For example, a user-level POSIX thread (pthread) in Linux will typically allocate 8MB of stack. In this scenario, each syscall thread imposes less than 0.2% overhead of memory per user-level pthread; this percentage decreases further when also considering the user-level heap of the application.

Despite spawning multiple threads, only *one* syscall thread is active per application and core at any given point in time. If system calls do not block all the work is executed by a single syscall

thread, while the remaining ones sleep on a work-queue. When a syscall thread needs to block, for whatever reason, immediately before it is put to sleep, FlexSC notifies the work-queue. Another syscall thread wakes-up and immediately starts executing the next system call. Later, when resources become free, current Linux code wakes up the waiting thread (in our case, a syscall thread), and resumes its execution, so it can post its result to the syscall page and return to wait in the FlexSC work-queue.

4.4.5 FlexSC Syscall Thread Scheduler

FlexSC implements a syscall thread scheduler that is responsible for determining when and on which core system calls will execute. This scheduler is critical to performance, as it influences the locality of user and kernel execution.

On a single-core environment, the FlexSC scheduler assumes the user-space will attempt to post as many exception-less system calls as possible, and subsequently call `flexsc_wait()`. The FlexSC scheduler then wakes up an available syscall thread that starts executing the first system call. If the system call does not block, the same syscall thread continues to execute the next submitted syscall entry. If the execution of a syscall thread blocks, the currently scheduled syscall thread notifies the scheduler to wake another thread to continue to execute more system calls. The scheduler does not wake up the user-space thread until all available system calls have been issued, and have either finished or are currently blocked with at least one system call having been completed. This is done to minimize the number of mode switches to user-space.

For multicore execution, the scheduler biases execution of syscall threads on a subset of available cores, dynamically specializing cores according to the workload requirements. The goal is to automatically increase and decrease the number of cores dedicated to operating system work depending on the number of system calls being generated by the application. An example of an application executing with FlexSC on a 4 core system is shown in Figure 4.8.

In our implementation of FlexSC, we use two triggers to re-evaluate whether the division of cores for kernel and application execution is correct. First, we use the `flexsc_wait()` call as an indication that the rate of completing operating system work may be insufficient, since `flexsc_wait()` is expected to be called by application threads when no more work can be performed without system call results. We further evaluate this assumption by verifying how many syscall pages contain requests that have been submitted and are not currently in progress. If there are more than 2 full syscall pages worth of calls (128 entries in our implementation), we decide to increase the number of cores dedicated to execute operating system work.

The second event that is used as a trigger to re-evaluate core allocation is when a syscall thread goes to sleep due to lack of work. If a syscall thread cannot find more work to do in any of the registered syscall pages for the process, it signifies that the application is not keeping up with the operating system. In this case, we decide to switch one of the cores that is currently dedicated to operating system execution, to execute application threads.

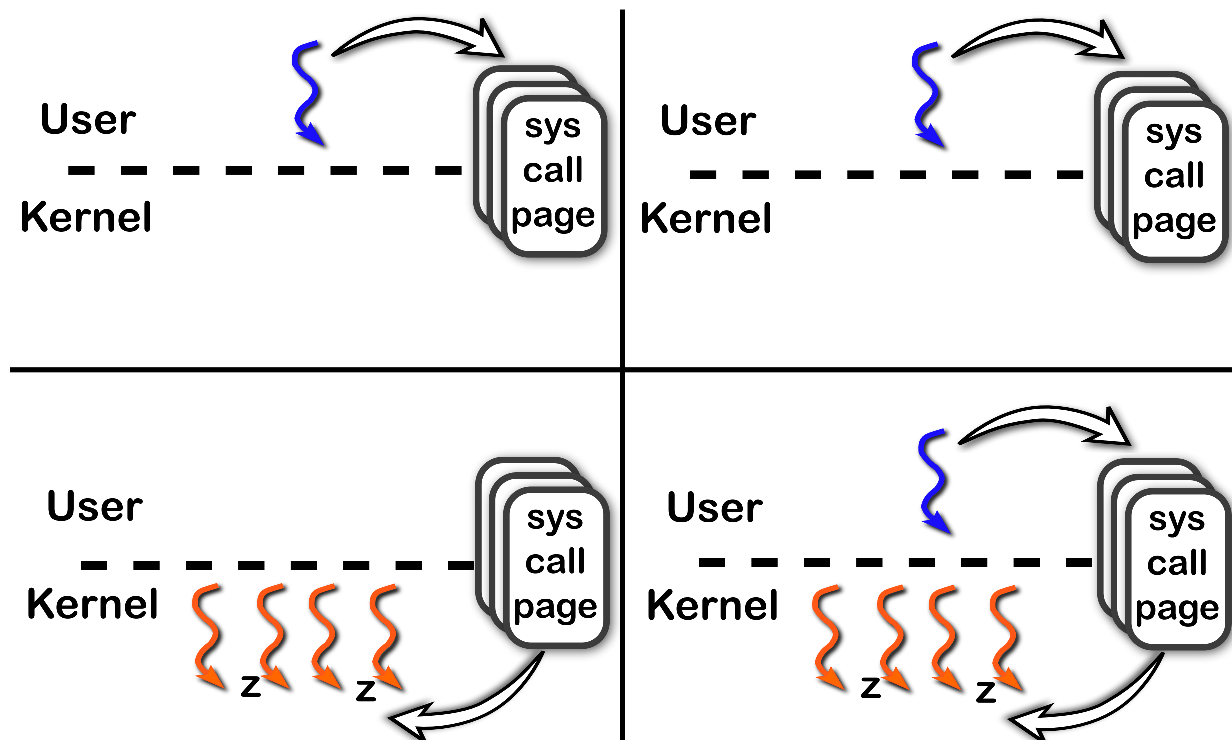


Figure 4.8: Example of FlexSC execution on a multicore system. In this diagram, we depict 4 cores running a single workload. The syscall pages, although depicted on all four cores, represent the *same* set of pages and are shared among all cores, leveraging the fast on-chip communication of multicores. On the top two cores, the operating system scheduler has scheduled user-level threads. These two cores are expected to maintain user-mode state in their caches, and asynchronously schedule requests that are written to syscall pages. On the bottom-left core, only syscall threads have been scheduled, as such, that core is expected to mostly contain kernel state in its structures. Finally, it is still possible for cores to time-share user and kernel execution, as is shown in the bottom-right core.

As previously described, there is never more than one syscall thread concurrently executing per core, for a given process. However in the multicore case, for the same process, there can be as many syscall threads as cores concurrently executing on the entire system. To avoid cache-line contention of syscall pages among cores, before a syscall thread begins executing calls from a syscall page, it *locks* the page until all its submitted calls have been issued. Since FlexSC processes typically map multiple syscall pages, each core on the system can schedule a syscall thread to work independently, executing calls from different syscall pages.

4.5 Summary

In this chapter, we identified and quantified sources of performance overhead associated with the system call mechanism that has been universally adopted by operating systems today. The sources of overhead can be classified into two components: a *direct* and an *indirect* component. The direct component, which has been traditionally measured by operating system micro-benchmarks, is tied to the cost of mode switching through the use a processor interrupt or exception. The indirect costs, which are rarely measured, relate to the processor state pollution that occurs with frequent mode switches, and prevents both application and kernel code from achieving peak efficiency throughout the execution of several thousand instructions.

Motivated by the performance cost analysis of the exception-based system call mechanism, we introduced the concept of exception-less system call that decouples system call invocation from execution. This allows for flexible scheduling of system call execution which in turn enables system call batching and dynamic core specialization that both improve locality in a significant way. System calls are issued by writing kernel requests to one of the reserved syscall pages using normal store operations, and they are executed by special in-kernel syscall threads, which then post the results to the syscall page.

The concept of exception-less system calls originated as a mechanism for low-latency communication between user and kernel-space with hyper-threaded processors in mind. We had hoped that communicating directly through the shared L1 cache would be much more efficient than relying on costly mode switching based mechanisms. However, the measurements presented in Section 4.2, as well as other preliminary experiments, made it clear that mixing user and kernel-mode execution on the same core would not be efficient for server class workloads. In future work we intend to study how to exploit exception-less system calls as a communication mechanism in hyper-threaded processors.

The implementation of this new mechanism, however, does not automatically result in performance improvements. In order to derive more efficient execution with exception-less system calls, it is also necessary for applications to make use of the new mechanism in a way that favors independent user and kernel execution. In the next two chapters, we explore different program structures that are able to exploit the performance benefits of exception-less system calls. In both

cases, we rely on highly parallel execution with sufficient independent work to overlap operating system and application execution.

Chapter 5

Exception-Less Threads

Exception-less system calls present a significant change to the semantics of the system call interface with potentially drastic implications for application code and programmers. Programming using exception-less system calls directly is more complex than using synchronous system calls, as they do not provide the same, easy-to-reason-about sequentiality. In addition, to reap the performance benefits of decoupling application and operating system execution, applications must exhibit inherent parallelism. Parallelism is necessary so that applications are able to continue executing useful work while waiting for replies from the operating system.

In this chapter, we address both of these challenges through a specialized threading library that, using exception-less system calls, can support existing multi-threaded applications. We describe the design and implementation of FlexSC-Threads, a new M-on-N threading package (M user-mode threads executing on N kernel-visible threads, with $M >> N$). The main purpose of this threading package is to harvest independent system calls from the application by switching threads, in user-mode, whenever an application thread invokes a system call.

We have implemented FlexSC-Threads so that it is binary compatible with POSIX threads, translating legacy synchronous system calls into exception-less ones *transparently* to applications. We show how FlexSC-Threads, with exception-less system call support, improves performance of important applications significantly. Specifically, FlexSC-Threads improves the performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 78% while requiring no modifications to the applications.

5.1 FlexSC-Threads Overview

Multiprocessing has become the default for computation on servers. With the emergence and ubiquity of multicore processors, along with projection of future chip manufacturing technologies, it is unlikely that this trend will reverse in the medium future. For this reason, and because of its relative simplicity vis-a-vis event-based programming, we believe that the multithreading concurrency model will continue to be a popular option for designing server applications.

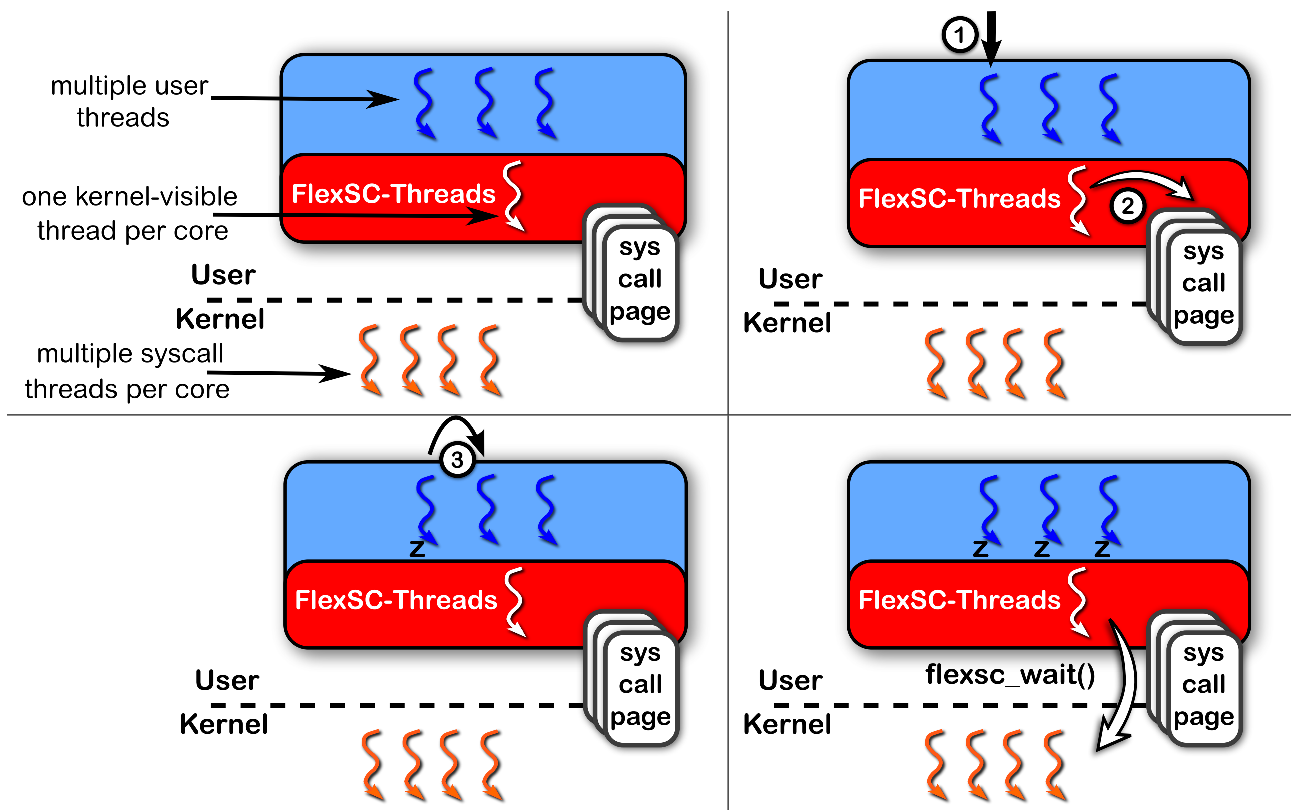


Figure 5.1: Illustration of FlexSC-Threads interacting with FlexSC on a single core; each of the four diagrams represents a different stage of execution. The top-left diagram depicts the components of FlexSC-Threads pertaining to a single core. The core run a single pinned kernel-visible thread, which in turn can multiplex multiple user-mode threads. Multiple syscall pages, and consequently syscall threads, are also allocated (and pinned) per core. The top-right diagram shows the FlexSC-Threads redirecting a system call request by, instead of issuing a special instruction, writing the request to an entry in the syscall page. The bottom-left diagram depicts a user-mode thread being preempted as a result of issuing a system call, and execution redirected to a ready user thread. This user-mode thread switch does not require interaction with the operating system kernel. The bottom-right diagram depicts the scenario where all user-mode threads are waiting for system call requests; in this case FlexSC-Threads library synchronously invokes `flexsc_wait()` to the kernel.

In this chapter, we describe the design and implementation of **FlexSC-Threads**, a threading package that transforms legacy synchronous system calls into exception-less ones *transparently* to applications. It is intended for server-type applications with many user-mode threads, such as Apache or MySQL. FlexSC-Threads is compliant with POSIX Threads, and binary compatible with NPTL [65], the default Linux thread library. As a result, Linux multi-threaded programs work with FlexSC-Threads "out of the box" without modification or recompilation.

FlexSC-Threads uses a simple M-on-N threading model (M user-mode threads executing on N kernel-visible threads). We rely on the ability to perform user-mode thread switching solely in user-space to transparently transform legacy synchronous calls into exception-less ones. In essence, FlexSC-Threads uses each kernel visible execution context (kernel-visible thread) to multiplex the execution of various user-mode threads, as shown in the top-left diagram of Figure 5.1.

For each newly created user-mode thread, a new stack is allocated, along with metadata structures that describe the thread. Each kernel-visible thread under control of the FlexSC-Threads library has a run queue of user-mode threads that are ready to execute, as well as a queue of user-mode threads that are blocked, waiting for a resource to become available.

As a consequence, FlexSC-Threads also implements a scheduler to schedule the execution of the user-level threads. Similar to traditional threading libraries, synchronizations points, such as locks, semaphores and access to contended resources form natural scheduling points. Unique to FlexSC-Threads is the use of system calls as an important scheduling point.

A fundamental feature of FlexSC-Threads is the use of user-mode thread switches, upon a system call, to hide the asynchronous nature of exception-less system calls. When execution of a user-mode thread results in a system call, the thread is blocked, a thread switch is made, and execution resumes. Only when the system call has been completed is the requesting thread allowed to resume execution.

The result of blocking threads while their respective exception-less system calls are pending is that system calls retain their legacy synchronous behavior. The result of performing user-mode thread switches upon system calls, on the other hand, is that the underlying implementation of the system call need not be synchronous, and in fact, is asynchronous with exception-less system calls. The expected outcome of such execution is that the number of mode switches can be substantially reduced. Moreover, user-mode threads and syscall threads, in the kernel, can work independently while asynchronously communicating through memory.

The interaction between FlexSC-Threads and the exception-less system call mechanism is illustrated in Figure 5.1. From an implementation perspective, the figure illustrates the following steps:

1. We redirect to our FlexSC-Threads library each *libc* call that issues a legacy system call. Typically, applications do not directly embed code to issue system calls, but instead call wrappers in the dynamically loaded *libc*. We use the dynamic loading capabilities of Linux to redirect execution of such calls to our library.

2. The FlexSC-Threads library then posts the corresponding exception-less system call to a syscall page, marks the current thread as blocked, and switches to another user-mode thread that is ready for execution.

3. If all user-mode threads are marked as blocked, depleting the ready run-queue, the FlexSC-Threads library checks the syscall page for any syscall entries that have been completed, waking up the appropriate user-mode thread so that it can obtain the result of the completed system call.

4. As a last resort, if all user-mode threads are blocked and no user-mode thread can be woken up, `flexsc_wait()` is called, putting the kernel visible thread to sleep until one of the pending

system calls has completed.

The policy implemented in the FlexSC-Threads scheduler is the simple first-in-first-out policy. Threads are placed in the run-queue in the order in which their blocking conditions are satisfied; we expect that in most cases threads are blocked waiting for the completion of a system call.

5.2 Multi-Processor Support

The implementation of FlexSC-Threads for a single processor is relatively simple, as described in the previous section. To enable efficient execution on multiple processors, with particular emphasis on multicore execution, several changes were introduced which we describe in this section.

The use of M-on-N threading model used in FlexSC-Threads was partially motivated for efficient multicore support, where N (the number of kernel visible threads) is equal to the number of cores available to the application. The key challenge of previously proposed M-on-N threading models, such as scheduler activations [7], is guaranteeing that the kernel visible threads do not block. Blocking a kernel visible thread is detrimental to performance in a M-on-N model, as it prevents further execution of the user-level threads that are ready to run on that kernel visible thread. The solution adopted for scheduler activations was to *upcall* user-space, giving it a new execution stack/context whenever an event caused the original system call to block. FlexSC-Threads sidesteps this issue with exception-less system calls. Since the kernel visible threads of FlexSC-Threads do not block on system calls, there is no need to implement kernel *upcalls*.

One disadvantage FlexSC-Threads has relative to scheduler activations is scheduler activations' ability to control implicit blocking events, such as user-level page-faults. On the other hand, our experience with server workloads has shown that user-level *blocking* page-faults are not a frequent event, except during early initialization of the server process. Kernel page-faults are not problematic, as they only block one of the syscall threads. As for non-blocking user page-faults (also known as "in-core" page faults), FlexSC-Threads behaves exactly the same way the default Linux 1-on-1 threading system does; an exception is taken synchronously, the page-fault is handled, and the thread is allowed to continue without blocking.[1]

5.2.1 Per core data structures and synchronization

FlexSC-Threads implements multicore support by creating a *single* kernel visible thread *per* core available to the process, and pinning each kernel visible thread to a specific core. From a pure functional perspective, we could schedule more than one kernel visible thread per core. However, since kernel-visible threads only block when there is no more available work, there is no need

[1] To allow for truly exception-less page-faults, the MMUs of modern processors would need to be extended to support an *exception-less page-fault* mechanism, allowing user-space to continue execution while avoiding the access to the faulted data (by, for example, switching to a separate user-mode thread), while posting the exception-less page-fault to the kernel. We are not aware of any processor that supports this, or equivalent, feature.

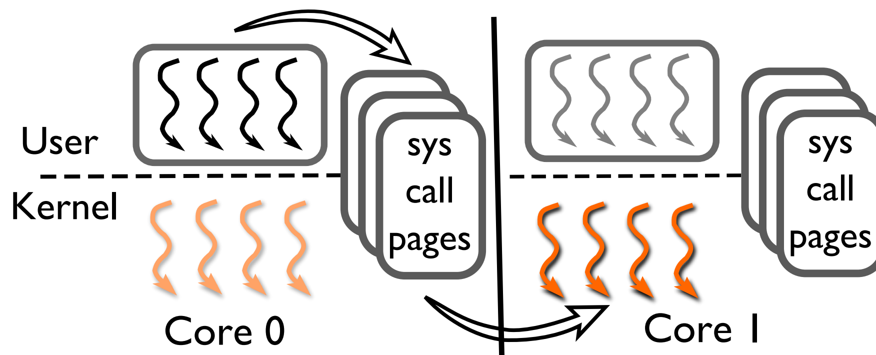


Figure 5.2: Simple 2 core example of FlexSC-Threads on FlexSC. Opaque threads are active, while grayed-out threads are inactive. Syscall pages are accessible to both cores, as we run using shared-memory, leveraging the fast on-chip communication of multicores.

to create more than one kernel visible thread per core. Furthermore, minimizing the number of context switches between kernel-visible FlexSC-Threads reduces the number of user/kernel mode switches.

Besides the benefits of reducing the number of mode switches, we leveraged the use of a single kernel visible thread per core to optimize FlexSC-Threads. In particular, we have structured most data structures in the library so that they are allocated and accessed per core (or kernel-visible thread). Operating on per core data structures has two performance advantages over using data structures that are shared by multiple cores. First, we expect that per core data structures can improve performance due to improved locality. Since these data structures are not accessed by multiple cores, they are not invalidated due to remote cores writing to shared cache lines. Second, redundancy across local caches is reduced, since local caches will contain mostly private data.

A second performance advantage of operating with per core data structures is that private data does not need to be protected with synchronization primitives, such as locks and atomic instructions. Since these primitives are quite expensive on current hardware, even when access is uncontended, avoiding them not only improves overall scalability, but also per core efficiency.

5.2.2 Thread migration

A couple of particularly important data structures in the FlexSC-Threads library are the run and wait queues, used by thread scheduler. For the performance reasons detailed in the previous section, it is important that the scheduler queues be per core. Given the targeted server style workload for FlexSC, where system calls are made once every few thousand instructions, thread switching is expected to be a significant portion of overhead that our library introduces.

One challenge of using per core wait and run queues when compared to central global queues is the problem of load imbalance. Load imbalance can lead to threads scheduled on some cores having less frequent access to the processor. Worst, in extreme cases where the run queue of a core is completely depleted, load imbalance can lead to cores experiencing idle time while ready to run

threads are available on run queues managed by other cores.

To resolve the potential for load imbalance in the FlexSC-Threads library, we implemented a simple thread migration protocol. The first component involved in keeping the run queues balanced is a public (visible to all cores) count of the current run queue length. Because of its advisory nature, the public run queue lengths are not synchronized for access. They are written only by their corresponding core and read (but never written to) by remote cores. These run queue lengths are used by each thread scheduler to determine whether they should request for a remote thread to be migrated to the local run queue.

In our implementation, the core with the least number of ready threads is responsible for initiating the migration protocol. It does so when the run queue with the most number of threads has more than two times the number of threads than the local queue. After deciding that a thread migration should occur, the local core signals the remote core that it wants to migrate one of the remote threads to the local run queue. To do so, it updates one of the data structures that is shared among the cores. Each core checks for migration requests upon executing the scheduler code. To satisfy the request, it removes the tail thread from the local run queue, and places it in the shared data structure (this data structure requires synchronization, unlike the per core private data structures). Finally, when the requesting core's scheduler executes again, it can finally take the migrated thread from the remote core and place it into its local run queue.

In addition to the traditional load imbalance challenge that exists in per core run queues, the FlexSC-Threads run-time has a unique situation that can lead to thread starvation. This occurs when, as described in Section 4.4.5, the syscall thread scheduler decides to schedule system call execution on a core. When it does so, it may specialize the core to predominantly execute kernel work, thus not allowing the kernel visible thread, pinned to only execute on that core, access to the processor. As a consequence, the user threads that were placed on the run queue of that particular kernel visible thread do get to run, and suffer from starvation.

To overcome this type of starvation, we extend the thread migration logic of the FlexSC-Threads scheduler to migrate user threads from a descheduled kernel visible thread to other cores that are currently executing in user-mode. Unlike migrating user threads between two kernel visible threads for the purpose of load balancing, moving user threads from a descheduled kernel visible thread cannot rely on the participation of the donor run queue (since its thread is descheduled). For this reason, the scheduler executing on remote cores must detect that a sibling kernel visible thread is not executing. We implement this by requiring the syscall thread scheduler to update a data structure, visible to FlexSC-Threads schedulers, whenever it begins executing syscall threads on a particular core. Inside the scheduler, cores can then monitor the status of remote cores. If a core determines that a remote core is executing in kernel mode, and the remote core has user threads pending in its run queue, it picks one thread from its run queue, with appropriate synchronization in place, and places it on the local run queue. If the core of the descheduled kernel visible thread stays in kernel mode for enough time, the sibling cores will eventually remove all of

the user threads from its run queue.

5.2.3 Syscall pages

As an optimization, we have designed FlexSC-Threads to register a private set of syscall pages *per* kernel visible thread (i.e., per core). Since syscall pages are private to each core, there is no need to synchronize their access with costly atomic instructions. The FlexSC-Threads user-mode scheduler implements a simple form of cooperative scheduling, with system calls acting as yield points. As a result, it is possible to guarantee atomicity of accesses to each syscall page entry without requiring synchronization.

It is important to note that FlexSC-Threads relies on a large number of independent user-mode threads to post concurrent exception-less system calls. Since threads are executing independently, there is no constraint on ordering or serialization of system call execution (thread-safety constraints should be enforced at the application level and is orthogonal to the system call execution model).

5.3 Limitations

Our implementation of FlexSC-Threads has been successfully used to run several server style workloads. We have been able to use the application binaries distributed with our Linux operating system, with small changes to startup scripts to enable the loading of our threading library instead of the default Linux threading library. Nonetheless, there are a few limitations of our current implementation of FlexSC-Threads that could potentially impact the applicability of our system to workloads we have not yet explored. Some of the limitations we are aware of are:

- Complications due to non-preemption. Our library does not implement preemption, that is, the ability to interrupt the execution of a user-mode thread at any moment. As a result, there is potential for a single user thread to execute indefinitely if it does not make a system call or attempt to acquire locks or mutexes, starving execution of other user threads. In all workloads we experimented with, lack of preemption did not pose a problem since during run-time threads frequently make system calls or synchronize with each other.

  A simple solution to implement preemption in user-level threading libraries is to schedule a periodic signal to be delivered to the user-level scheduler. This signal can be used to preempt the currently executing user thread. We believe such a mechanism could be adopted in FlexSC-Threads. The major change necessary would be ensuring atomicity of access to data structures that were previously unprotected due to the assumptions made about non-preemption.

- Idle time due to blocking page faults. As discussed in Section 5.2, if a user-level page-fault blocks the kernel visible thread (say, due to swapping), all of the user-level threads, even if ready to execute, are blocked from running as well. As mentioned, in our experiments we

have not found this to be an issue. Even when experimenting with MySQL, which is the most memory hungry server, most of the I/O is done through explicit system calls, as long as the memory buffers are sized correctly to fit in the available physical memory.

A potential solution for workloads that exhibit blocking page-faults is allowing multiple kernel visible threads to be scheduled per core (we currently allow only one). This is the traditional solution to avoiding idle time in 1-to-1 threading models (such as the current default Linux NPTL threading library [65]). In this case, when a kernel visible thread blocks, another ready kernel visible thread can be scheduled on the same processor, which prevents the processor from accruing idle time. The main drawback of having multiple kernel visible threads per core would be the increase in overhead due to the larger number of context switches.

- No multi application experiments. The experiments we conducted in the following sections are based on the assumption that a single application is running on the server. Although there is no fundamental design decision made based on this assumption, it is likely that we would have to refine the scheduling policies when multiple applications are competing for resources, specifically for execution on multiple processors. For example, the algorithm used to decide how many and which cores are dedicated to kernel-mode execution uses per process information (namely, number of outstanding system calls for that process). In the case of having multiple applications, it will likely be important to take into account information that is system wide, and not only per process.

  One solution that we have considered is to divide the number of cores available to different applications. Instead of time sharing the same cores, which is the strategy used on current operating systems, applications should be scheduled on a non-intersecting set of cores. After deciding how many cores to dedicate to which applications, the current FlexSC and FlexSC-Threads policies should work without requiring changes. However, deciding the optimal number of cores to dedicate per application, and potentially adapting that number at runtime, is still a topic of on-going research.

5.4 Experimental Evaluation

We first present the results of a micro-benchmark that shows the overhead of the basic exception-less system call mechanism, and then we show the performance of three popular server applications, Apache, MySQL, and BIND transparently using exception-less system calls through FlexSC-Threads. Finally, we analyze the sensitivity of the performance of FlexSC to the number of system call pages.

FlexSC was implemented in the Linux kernel, version 2.6.33. The baseline line measurements we present were collected using unmodified Linux (same version), and the default native POSIX threading library (NPTL) [65]. We identify the baseline configuration as "**sync**", and the system

Component	Specification
Cores	4
Issue width	5 instructions
Reorder Buffer	128 entries
Cache line	64 B for all caches
Private L1 i-cache	32 KB, 3 cycle latency
Private L1 d-cache	32 KB, 4 cycle latency
Private L2 cache	512 KB, 11 cycle latency
Shared L3 cache	8 MB, 35-40 cycle latency
Memory	250 cycle latency (avg.)
TLB (L1)	64 (data) + 64 (instr.) entries
TLB (L2)	512 entries

Table 5.1: Characteristics of the 2.3GHz Core i7 processor.

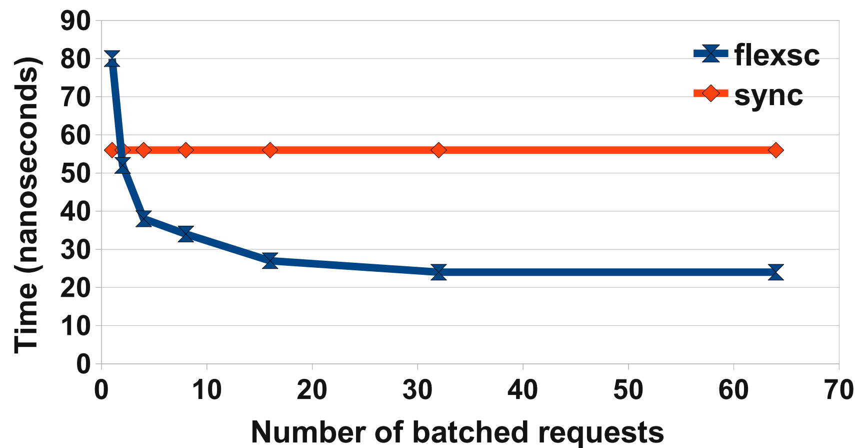


Figure 5.3: Exception-less system call cost on a single-core.

with exception-less system calls as “**flexsc**”.

The experiments presented in this section were run on an Intel Nehalem (Core i7) processor with the characteristics shown in Table 5.1. The processor has 4 cores, each with 2 hyper-threads. We disabled the hyper-threads, as well as the “TurboBoost” feature, for all our experiments to more easily analyze the measurements obtained.

For the Apache, MySQL, and BIND experiments, requests were generated by a remote client connected to our test machine through a 1 Gbps network, using a dedicated router. The client machine contained a dual core Core2 processor, running the same Linux installation as the test machine, and was not CPU or network constrained in any of the experiments.

All values reported in our evaluation represent the average of 5 separate runs.

5.4.1 Overhead

The overhead of executing an exception-less system call involves switching to a syscall thread, de-marshalling arguments from the appropriate syscall page entry, switching back to the user-thread,

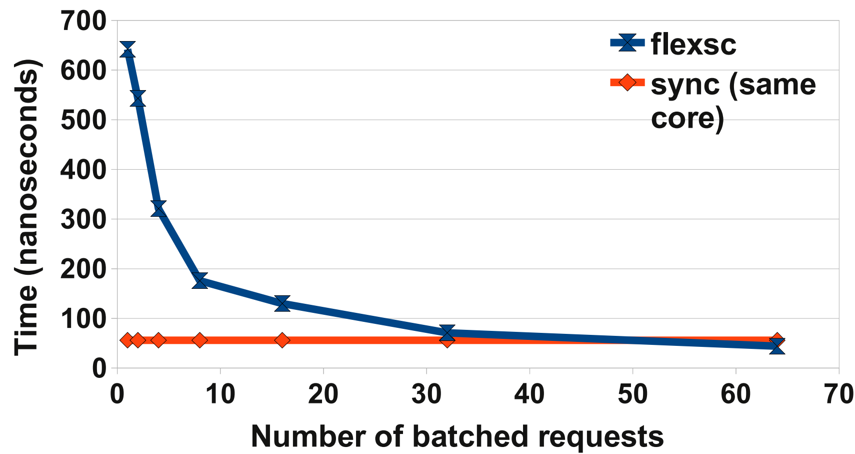


Figure 5.4: Exception-less system call cost, in the worst case, for remote core execution.

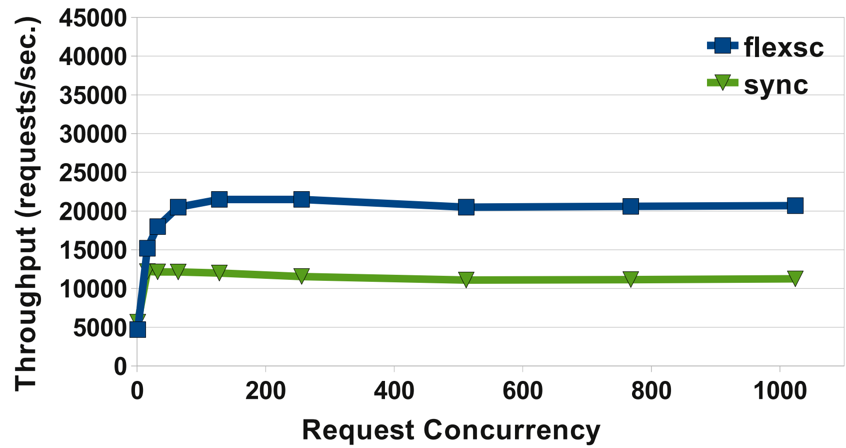
and retrieving the return value from the syscall page entry. To measure this overhead, we created a micro-benchmark that successively invokes a `getppid()` system call. Since the user and kernel footprints of this call are small, the time measured corresponds to the *direct* cost of issuing system calls.

We varied the number of batched system calls, in the exception-less case, to verify if the direct costs are amortized when batching an increasing number of calls. The results obtained executing on a single core are shown in Figure 5.3. The baseline time, show as a horizontal line, is the time to execute an exception-based system call on a single core. Executing a single exception-less system call on a single core is 43% slower than a synchronous call. However, when batching more than 2 calls the overhead is lower than for synchronous call, and when batching 32 or more calls, the execution of each call is up to 130% faster than a synchronous call.

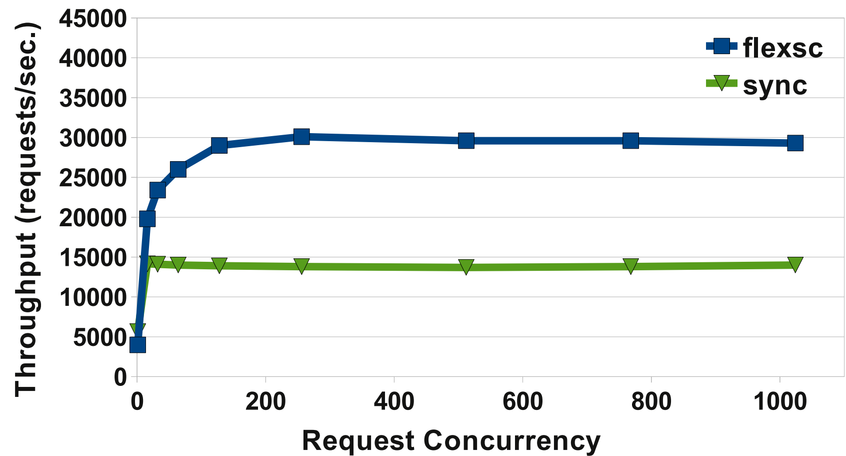
We also measured the time to execute system calls on a remote core (Figure 5.4). In addition to the single core operations, remote core execution entails sending an inter-processor interrupt (IPI) to wake up the remote syscall thread. In the remote core case, the time to issue a single exception-less system call can be more than 10 times slower than a synchronous system call on the same core. This measurement represents a worst case scenario when there is no currently executing syscall thread. Despite the high overhead, the overhead on remote core execution is recouped when batching 32 or more system calls.

5.4.2 Apache

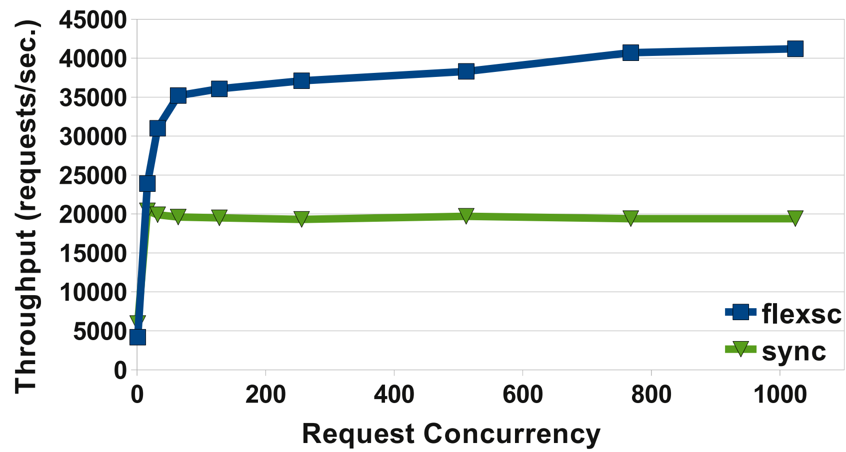
We used Apache version 2.2.15 to evaluate the performance of FlexSC-Threads. Since FlexSC-Threads is binary compatible with NPTL, we used the same Apache binary for both FlexSC and Linux/NPTL experiments. We configured Apache to use a different maximum number of spawned threads for each case. The performance of Apache running on NPTL degrades with too many threads, and we experimentally determined that 200 was optimal for our workload and hence



(a) 1 Core



(b) 2 Cores



(c) 4 Cores

Figure 5.5: Comparison of Apache throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.

used that configuration for the NPTL case. For the FlexSC-Threads case, we raised the maximum number of threads to 1000.

The workload we used was ApacheBench, a HTTP workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 5.5 shows the results of Apache running on 1, 2 and 4 cores. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically redirects system calls to maximize core locality. The results show that, except for a very low number of concurrent requests, FlexSC outperforms Linux/NPTL by a wide margin. With system call batching alone (1 core case), we observe a throughput improvement of up to 86%. The 2 and 4 core experiments show that FlexSC achieves up to 116% throughput improvement, showing the added benefit of dynamic core specialization.

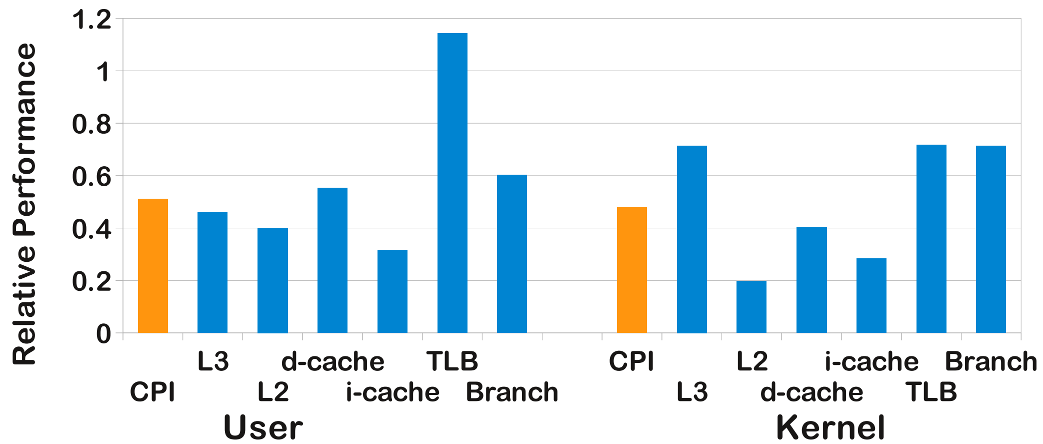
Table 5.2 and Figure 5.6 show the effects of FlexSC on the micro-architectural state of the processor while running Apache. They display various processor metrics, collected using hardware performance counters, during execution with 512 concurrent requests. The most important metric listed in Table 5.2 is the cycles per instruction (CPI) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution. The other values listed are normalized values using *misses per kilo-instructions* (MPKI). MPKI is a widely used normalization method that makes it easy to compare values obtained from different executions. Figure 5.6 is based on the same information, however, it displays the *relative* performance of each of the metrics, to allow for easier visual comparison between Linux/NPTL and FlexSC.

The most efficient execution of the four listed in the table is FlexSC on 1 core, yielding a CPI of 1.06 on both kernel and user execution, which is 95–108% more efficient than for NPTL. While the FlexSC execution of Apache on 4 cores is not as efficient as the single core case, with an average CPI of 1.33, there is still a 71% improvement, on average, over NPTL.

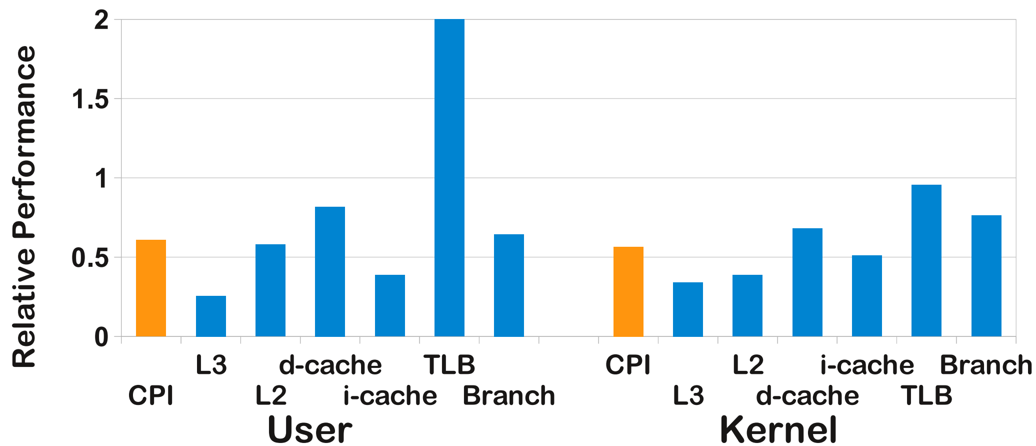
Most metrics we collected are significantly improved with FlexSC. Of particular importance are the performance critical structures that have a high MPKI value on NPTL such as d-cache, i-cache, and L2 cache. The better use of these micro-architectural structures effectively demonstrates the premise of this work, namely that exception-less system calls can improve processor efficiency. The only structure which observes more misses on FlexSC is the user-mode TLB. We are currently investigating the reason for this.

There is an interesting disparity between the throughput improvement (94%) and the CPI improvement (71%) in the 4 core case. The difference comes from the added benefit of localizing kernel execution with core specialization. Figure 5.7 shows the time breakdown of Apache executing on 4 cores. FlexSC execution yields significantly less idle time than the NPTL execution.[2] The reduced idle time is a consequence of lowering the contention on a specific kernel semaphore,

[2] The execution of Apache on 1 or 2 core did not present any idle time.



(a) 1 Core



(b) 4 Cores

Figure 5.6: Comparison of processor performance metrics of Apache execution using Linux and FlexSC on 1 and 4 cores. All values are normalized to baseline execution (**sync**). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

Apache	User							Kernel						
Setup	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch
sync (1 core)	**2.08**	3.7	68.9	63.8	130.8	7.7	20.9	**2.22**	1.4	80.0	78.2	159.6	4.6	15.7
flexsc (1 core)	**1.06**	1.7	27.5	35.3	41.3	8.8	12.6	**1.06**	1.0	15.8	31.6	45.2	3.3	11.2
sync (4 cores)	**2.22**	3.9	64.6	67.9	127.6	9.6	20.2	**2.32**	4.4	49.5	73.8	124.9	4.4	15.2
flexsc (4 cores)	**1.35**	1.0	37.5	55.5	49.4	19.3	13.0	**1.31**	1.5	19.1	50.2	63.7	4.2	11.6

Table 5.2: Micro-architectural breakdown of Apache execution on uni- and quad-core setups. All values shown, except for CPI, are normalized using misses per kilo-instruction (MPKI): therefore, lower values yield more efficient execution and lower CPI.

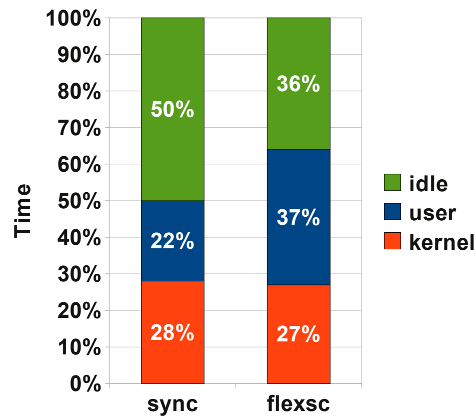


Figure 5.7: Breakdown of execution time into kernel, user and idle time of the Apache workload on 4 cores.

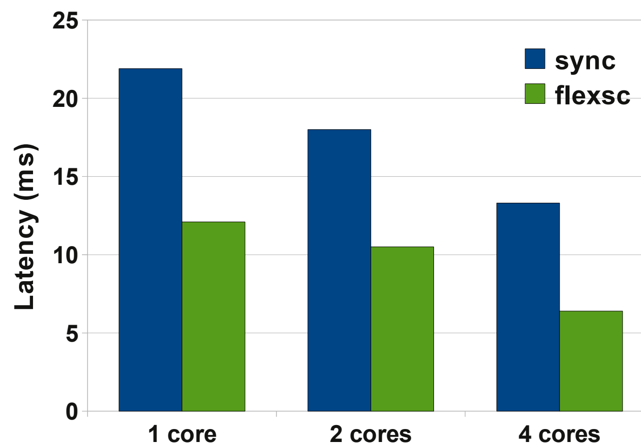


Figure 5.8: Comparison of Apache latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

namely `mmap_sem`. Linux protects address spaces with a per address-space read-write semaphore (`mmap_sem`). Profiling shows that every Apache thread allocates and frees memory for serving requests, and both of these operations require the `mmap_sem` semaphore to be held with write permission. Further, the network code in Linux invokes `copy__user()`, which transfers data in and out of the user address-space. This function verifies that the user-space memory is indeed valid, and to do so acquires the semaphore with read permissions. In the NPTL case, threads from all 4 cores compete on this semaphore, resulting in 50% idle time. With FlexSC, kernel code is dynamically scheduled to run predominantly on 2 out of the 4 cores, halving the contention to this resource, eliminating 38% of the original idle time.

Another important metric for servicing Web requests besides throughput is latency of individual requests. One might intuitively expect that latency of requests to be higher under FlexSC because of batching and asynchronous servicing of system calls, but the opposite is the case. Figure 5.8 shows the average latency of requests when processing 256 concurrent requests (other concurrency levels showed similar trends). The results show that Web requests on FlexSC are serviced within 50-60% of the time needed on NPTL, on average.

5.4.3 MySQL

In the previous section, we demonstrated the effectiveness of FlexSC running on a workload with a significant proportion of kernel time. In this section, we experiment with online transaction processing (OLTP) workload on MySQL, a workload for which the proportion of kernel execution is smaller (roughly 25% as seen in Figure 5.9). Our evaluation used MySQL version 5.5.4 with an InnoDB backend engine, and as in the Apache evaluation, we used the same binary for running on NPTL and on FlexSC. We also used the same configuration parameters for both the NPTL and FlexSC experiments, after tuning them for the best NPTL performance.

To generate requests to MySQL, we used the *sysbench* system benchmark utility. Sysbench was

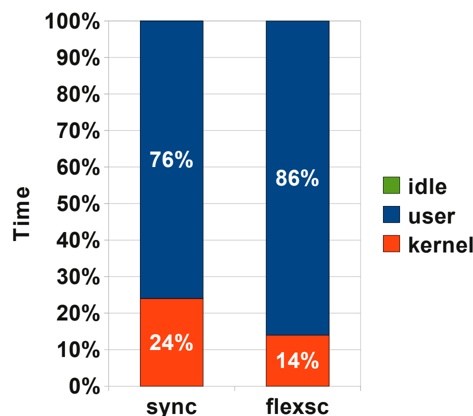
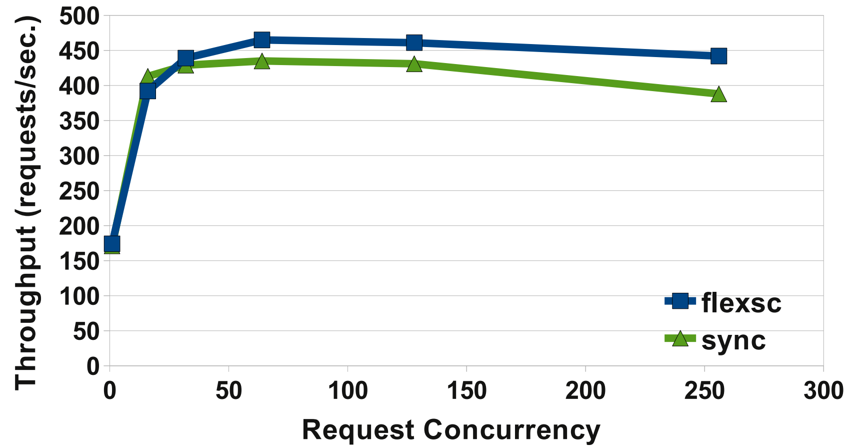
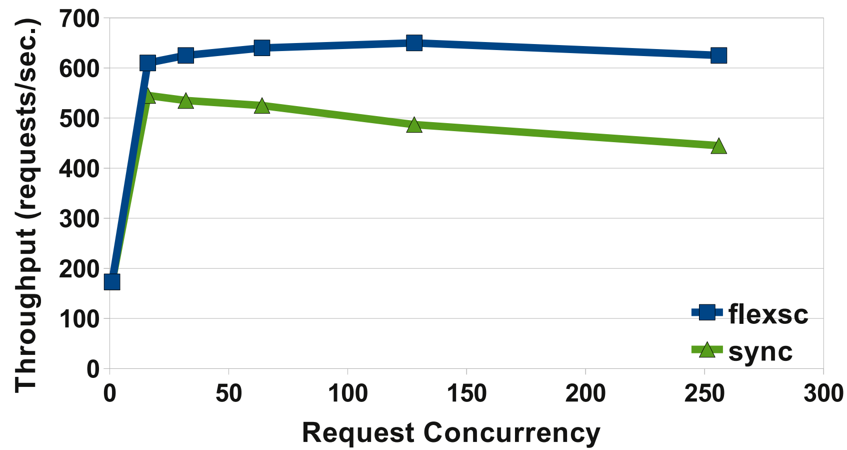


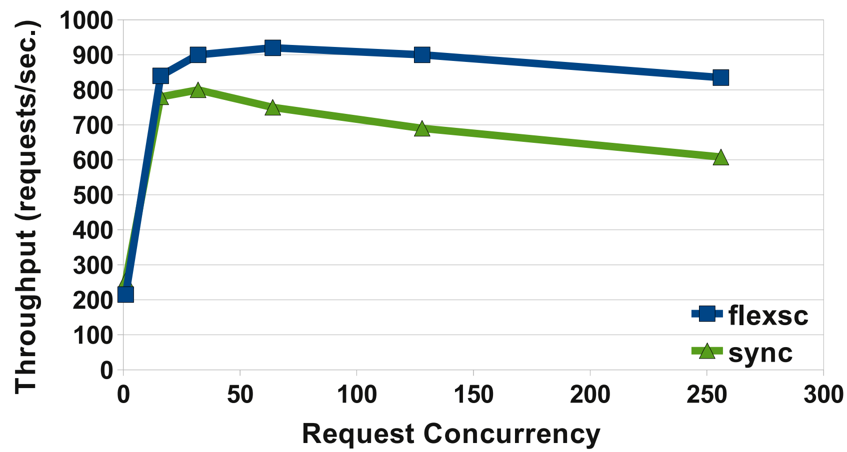
Figure 5.9: Breakdown of execution time into kernel, user and idle time of the MySQL workload on 4 cores.



(a) 1 Core

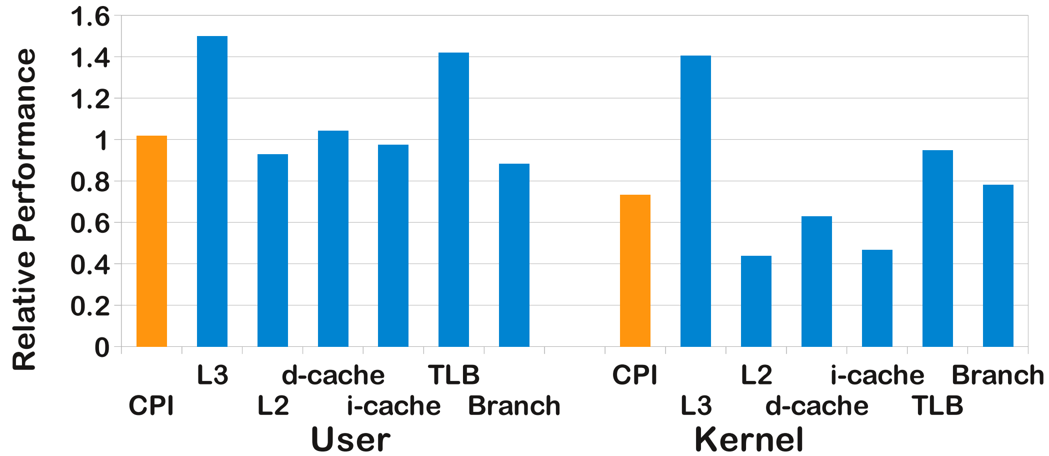


(b) 2 Cores

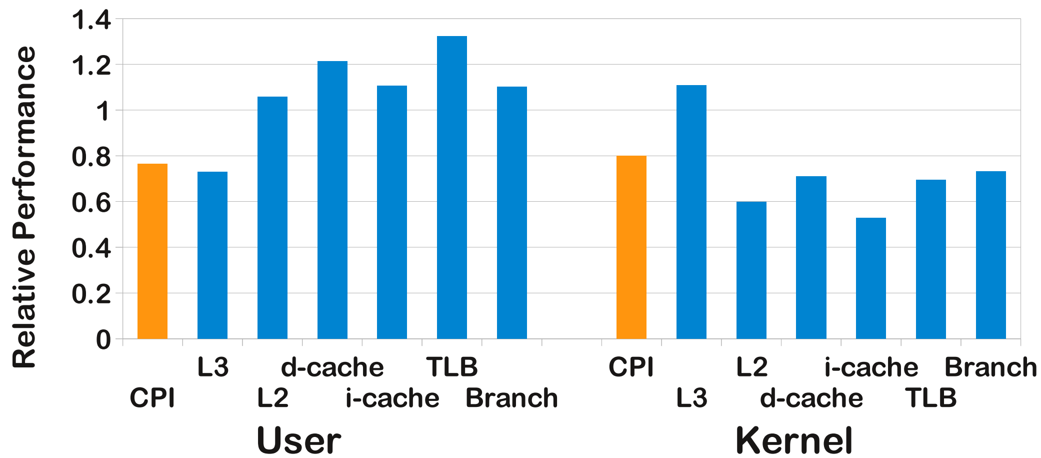


(c) 4 Cores

Figure 5.10: Comparison of MySQL throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.



(a) 1 Core



(b) 4 Cores

Figure 5.11: Comparison of processor performance metrics of MySQL execution using Linux and FlexSC on 1 and 4 cores. All values are normalized to baseline execution (**sync**). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

MySQL	User							Kernel						
Setup	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch
sync (1 core)	**0.89**	0.6	21.1	34.8	24.2	3.8	7.8	**3.03**	16.5	125.2	209.6	184.9	3.9	17.4
flexsc (1 core)	**0.91**	1.8	19.6	36.3	23.6	5.4	6.9	**2.22**	23.2	55.1	131.9	86.5	3.7	13.6
sync (4 cores)	**1.82**	3.7	15.8	25.2	18.9	3.1	5.9	**2.78**	16.6	78.0	147.0	120.0	3.6	15.7
flexsc (4 cores)	**1.39**	2.7	16.7	30.6	20.9	4.7	6.5	**2.22**	18.4	46.6	104.4	63.5	2.5	11.5

Table 5.3: Micro-architectural breakdown of MySQL execution on uni- and quad-core setups. All values shown, except for CPI, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and lower CPI.

created for benchmarking MySQL processor performance and contains an OLTP inspired workload generator. The benchmark allows executing concurrent requests by spawning multiple client threads, with each connecting to the server and sequentially issuing SQL queries. The number of concurrent connections (concurrency level) can be configured for each experiment. To handle the concurrent clients, MySQL spawns a user-level thread per connection. At the end, sysbench reports the number of transactions per second executed by the database, as well as average latency information. For these experiments, we used a database with 5M rows, resulting in 1.2 GB of data. Since we were interested in stressing the CPU component of MySQL, we disabled synchronous transactions to disk. Given that the configured database was small enough to fit in memory, the workload presented no idle time due to disk I/O.

Figure 5.10 shows the throughput numbers obtained on 1, 2 and 4 cores when varying the number of concurrent client threads issuing requests to the MySQL server.[3] For this workload, system call batching on one core provides modest improvements: up to 14% with 256 concurrent requests. On 2 and 4 cores, however, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 37%-40% higher throughput.

Table 5.3 contains the micro-architectural processor metrics collected for the execution of MySQL. For easier visualization, we compare the metrics between NPTL and FlexSC in Figure 5.11. Because MySQL invokes the kernel less frequently than Apache, kernel execution yields high miss rates, resulting in a high CPI of 3.03 on NPTL. In the single core case, FlexSC does not greatly alter the execution of user-space, but decreases kernel CPI by 36%. FlexSC allows the kernel to reuse state in the processor structures, yielding lower misses across most metrics. In the case of 4 cores, FlexSC also improves the performance of user-space CPI by as much as 30%, compared to NPTL. Despite making less of an impact in the kernel CPI than in single core execution, there is still a 25% kernel CPI improvement over NPTL.

Figure 5.12 shows the average latencies of individual requests for MySQL execution with 256 concurrent clients. As is the case with Apache, the latency of requests on FlexSC is improved over execution on NPTL. Requests on FlexSC are satisfied within 70-88% of the time used by requests on NPTL.

[3]For both NPTL and FlexSC, increasing the load on MySQL yields peak throughput between 32 and 128 concurrent clients after which throughput degrades. The main reason for this performance degradation is the costly and coarse synchronization used in MySQL. MySQL and Linux kernel developers have observed similar performance degradation.

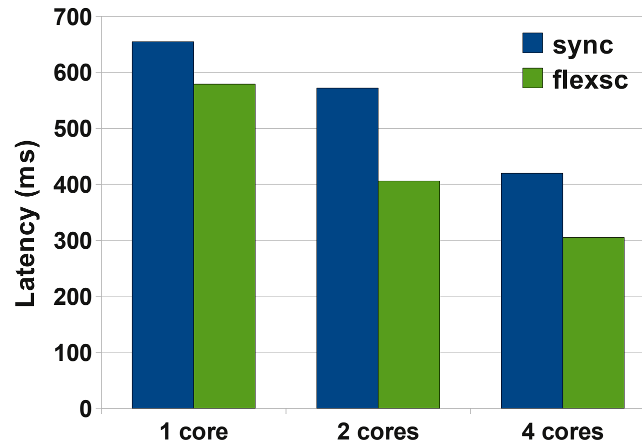


Figure 5.12: Comparison of MySQL latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

5.4.4 BIND

The Domain Name System (DNS) is a vital component of modern networks, including the Internet. At a high level, DNS is used to translate names, which are meant to be human-friendly, into numeric addresses, which are used for identification of computers and IP packet routing across networks. In this section, we evaluate the performance of a DNS server under FlexSC. In particular, we use the BIND 9 DNS server, which was developed by the Internet Software Consortium as the de facto standard DNS server.

In the experiments presented we use BIND version 9.7.0, and as with the previous servers, we use the same binary and installation to generate both NPTL on FlexSC results. We configured BIND to be an authoritative server for four local (made up) zones. Through experimentation, we observed best performance when running BIND with 64 threads per core, in both NPTL and FlexSC-Threads.

The workload we used to generate DNS requests was the *dnsperf* DNS server performance tool (version 1.0.1.0). It uses an input file containing queries to be issued to the server. The tool generates requests at an increasing frequency until the server achieves peak throughput. As such, the concurrency level cannot be controlled and the tool reports the overall throughput (in transactions per second), along with average and maximum latencies for the requests. For the experiments presented in this section, we generated an input file with 500 thousand DNS requests, based on an example input file that is distributed with *dnsperf*.

With respect to the proportion of execution time in user and kernel modes, BIND has higher proportion of user mode execution than kernel mode execution, as can be seen in Figure 5.13. On a single core, when using NPTL threading, BIND spends close to one-third of the time executing in kernel-mode, which is a higher proportion of kernel time relative to MySQL, but less than Apache.

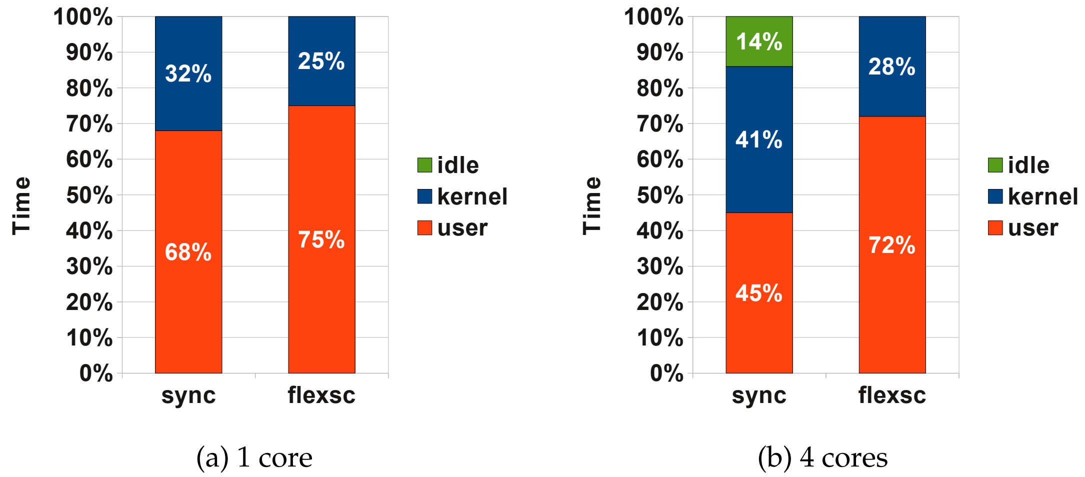


Figure 5.13: Breakdown of execution time into kernel, user and idle time of the BIND workload on 1 and 4 cores.

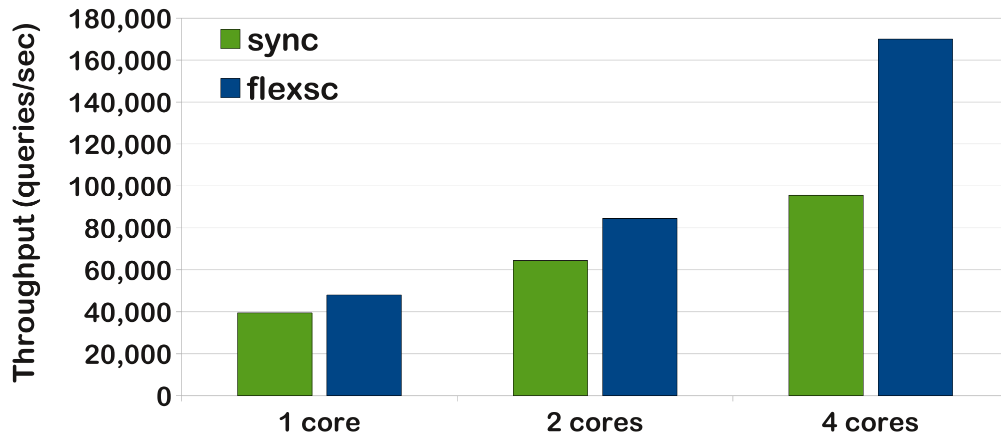


Figure 5.14: Comparison of BIND throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.

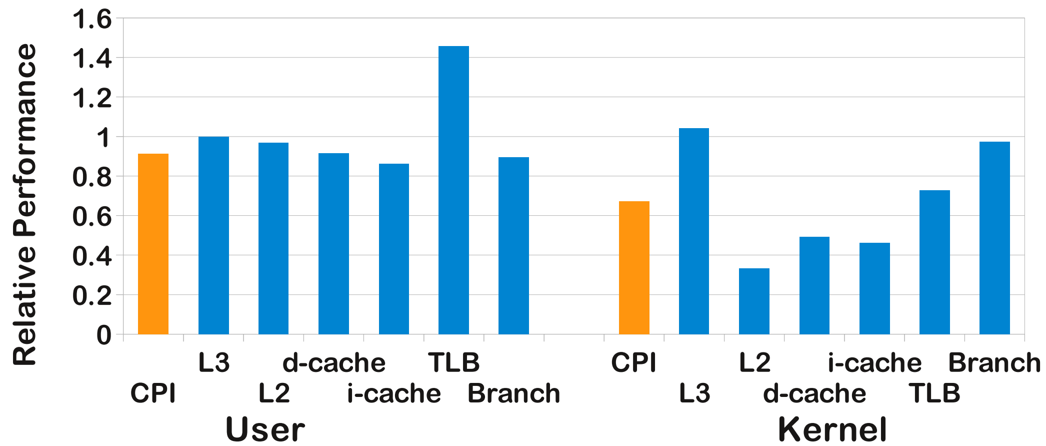
Figure 5.14 shows the throughput of BIND executing on 1, 2 and 4 cores. For BIND, system call batching on one core provides an improvement of 22% in the throughput reported by *dnsperf*. The performance improvement of FlexSC over NPTL is higher when executing with 2 cores, with a 31% throughput increase. With 4 cores, we observe the highest performance difference between FlexSC and NPTL, with up to 79% throughput increase.

The results show that not only does FlexSC yield faster execution, but it scales better with the number of cores. With NPTL, each doubling in the number of processors used to execute BIND, we observe a 1.55 times increase in the peak throughput. With FlexSC, each doubling of processors yields a roughly 1.85 times increase in the the peak throughput, bringing scalability closer to ideal (which would be a 2 times increase in throughput with each doubling in core count).

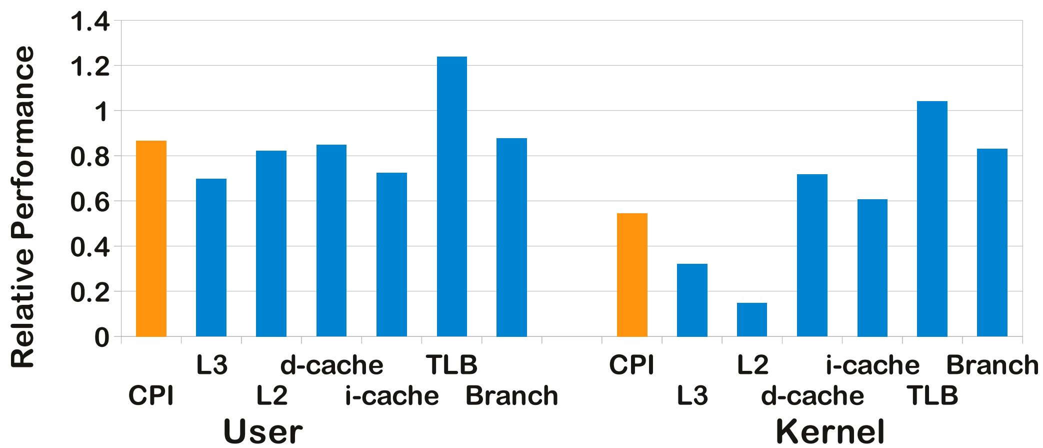
Table 5.4 and Figure 5.15 show the effects of FlexSC on the micro-architectural state of the processor while running BIND. Similar to MySQL, which is dominated by user-mode execution, the architectural improvements are greatest within kernel-mode execution. Since BIND execution is biased towards user-mode, separating user and kernel execution yields modest improvements to user-mode: 10% on 1 core and 19% on 4 cores. For kernel-mode execution, however, FlexSC improves efficiency by 49% in 1 core and on 4 cores, efficiency improves by up to 75%. As the micro-architectural breakdown shows, several processor structures that are important for performance such as the L1 *i-cache*, L1 *d-cache* and L2 saw reduced miss rate when BIND executed with FlexSC. The reductions are particularly visible for kernel-mode execution on 4 cores, where L2 misses were reduced to less than one-third of their baseline values and L1 icache misses were reduced to less than half of the baseline values.

In addition to the improvements in processor efficiency, the execution of BIND also benefits from reduced synchronization costs in FlexSC due to localized kernel execution. Analogous to the idle time observed with Apache when executing on 4 cores, BIND exhibits 14% idle time with Linux/NPTL when running on 4 cores, as can be seen in Figure 5.13. The source of the idle time is contention for a mutex that protects sockets (`sk_lock`). Since the *dnsperf* tool opens a single UDP connection to the BIND server, all BIND threads must acquire the same lock to send and receive messages. In the 4 core execution with FlexSC, most operating system activity is localized to a single core. For this reason, the `sk_lock` lock is predominantly accessed from a single core, which significantly reduces the contention for the lock. As a result, execution with FlexSC on 4 cores exhibits no idle time.

Figure 5.16 shows the latencies of requests as reported by the *dnsperf* tool. The left graph shows the *average* latencies for requests and the right graph shows the *maximum* latencies for requests. For the average case, FlexSC reduces the latency proportionally to the increases in throughput: 19% on 1 core, 25% on 2 cores and 43% on 4 cores. For the maximum case, FlexSC reduces latencies even further, by as much as 85% on 1 core, and more than 72% on 2 and 4 cores.



(a) 1 Core

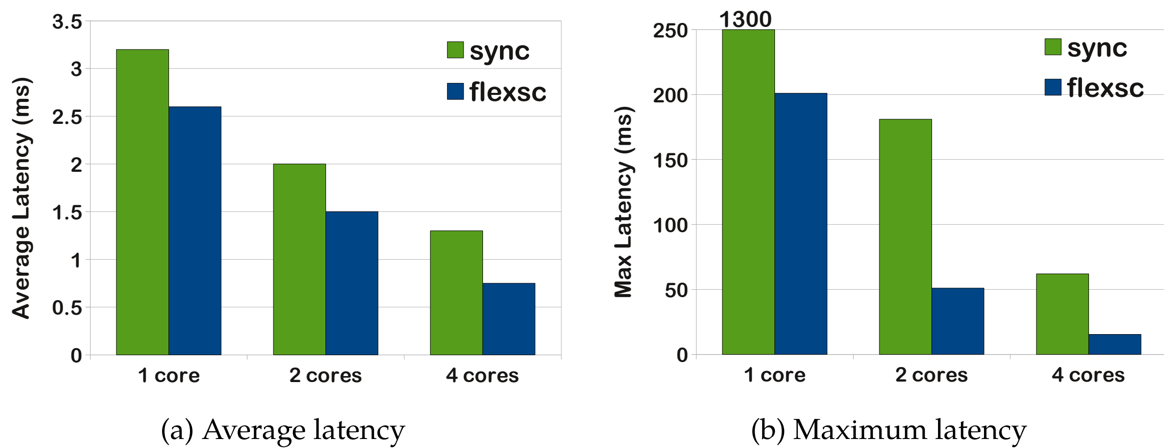


(b) 4 Cores

Figure 5.15: Comparison of processor performance metrics of BIND execution using Linux and FlexSC on 1 and 4 cores. All values are normalized to baseline execution (**sync**). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

MySQL	User							Kernel						
Setup	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch
sync (1 core)	**1.24**	0.2	18.0	76.8	99.2	6.4	20.0	**2.06**	0.7	30.4	150.0	210.5	9.0	10.0
flexsc (1 core)	**1.13**	0.2	17.5	70.4	85.6	9.2	18.0	**1.39**	0.7	10.1	73.8	97.2	6.5	9.8
sync (4 cores)	**1.48**	2.2	24.2	76.1	100.6	7.4	21.2	**2.57**	6.4	37.2	95.7	135.0	6.5	12.1
flexsc (4 cores)	**1.24**	1.5	18.3	65.4	71.4	9.4	18.2	**1.46**	2.0	5.3	70.4	88.2	7.3	10.1

Table 5.4: Micro-architectural breakdown of BIND execution on uni- and quad-core setups. All values shown, except for CPI, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and lower CPI.



(a) Average latency

(b) Maximum latency

Figure 5.16: Comparison of BIND latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores. The left graph shows the *average* latencies reported by the *dnsperf* client, and the right graph shows the *maximum* latency taken by any single request.

5.4.5 Sensitivity Analysis

In all experiments presented so far, FlexSC was configured to have 8 system call pages per core, allowing up to 512 concurrent exception-less system calls per core. Figure 5.17 shows the sensitivity of FlexSC to the number of available syscall entries. It depicts the throughput of Apache, on 1 and 4 cores, while servicing 2048 concurrent requests per core, so that there would always be more requests available than syscall entries. Uni-core performance approaches its best with 200 to 250 syscall entries (3 to 4 syscall pages), while quad-core execution starts to plateau with 300 to 400 syscall entries (6 to 7 syscall pages).

It is particularly interesting to compare Figure 5.17 with Figure 5.3 and 5.4. The results obtained from micro-benchmarking system call invocation (Figure 5.3 and 5.4) would indicate that there is no need to have applications issue more than 32 exception-less syscall calls. Yet, Figure 5.4 shows that for Apache, there are clear performance benefits in using several times more syscall entries (almost 10 times more syscall entries).

We believe these results confirm the system call costs analysis made in Section 4.2. In the micro-benchmark, which mostly suffers from the direct costs of system calls (mode switches) performance reaches its peak with the reduction of a few processor exceptions. With the execution of a real application, such as Apache, we observe that for performance benefits are observed for longer periods of specialized (user or kernel) execution. This comparison is another indication that the direct cost of mode switching, has a lesser effect on performance when compared to the indirect cost of mixing user- and kernel-mode execution.

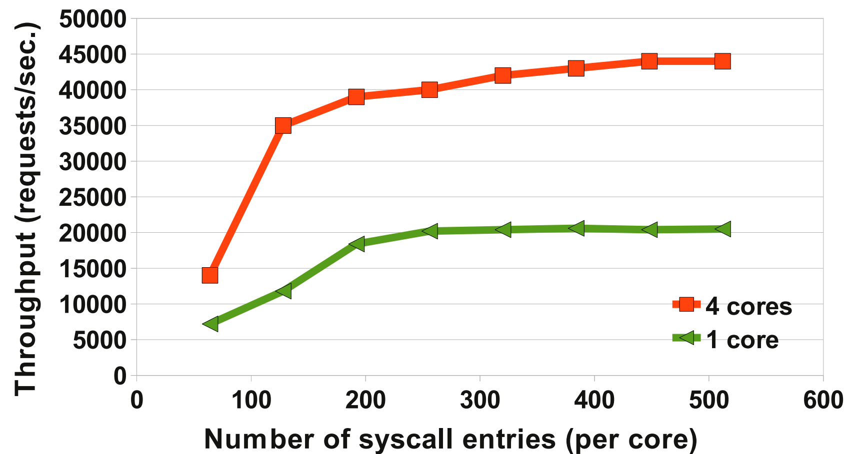


Figure 5.17: Execution of Apache on FlexSC-Threads, showing the performance sensitivity of FlexSC to different number of syscall pages. Each syscall page contains 64 syscall entries.

5.5 Discussion

5.5.1 Increase of user-mode TLB misses

The performance analysis of applications executing under FlexSC-Threads and FlexSC we presented in this chapter included detailed measurements of performance sensitive processor structures. We were able to verify that the application performance improvements observed were consistent with improvements in the cycles-per-instruction (CPI) processor metric, as well as improved execution locality demonstrated by more effective use of various processor structures. Among the performance sensitive processor structures stressed in the workloads we studied, the L1 instruction and data caches, which reached values of more than 100 misses per 1000 instructions, and the L2 caches exhibited improved utilization under FlexSC.

The only metric that was consistently worse under FlexSC, for most experiments, was the misses on the user-mode TLB. After extensive performance analysis, comparing TLB miss behavior under default Linux and FlexSC, we were unable to ascertain the cause of the increase in user-mode TLB misses.

One potential source of extra TLB misses are the accesses to the shared memory pages (syscall pages) that is unique to FlexSC. However, TLB miss profiling showed that the addresses that triggered TLB misses were not concentrated on the syscall pages. In fact, TLB misses were uniformly spread throughout application's address space, without a discernible pattern or bias to a specific region of the address space.

Profiling user-mode TLB misses indicated that a cause could be the finer granularity of switching user-mode threads in the FlexSC-Threads library compared to the Linux/NPTL scheduler. While Linux/NPTL scheduler only switches user threads upon a blocking event (such as I/O or a lock), FlexSC-threads performs a switch on every system call invocation. Since each user-level thread may access a distinct span of the address space, fine-grain switching of threads may put

added pressure on the TLB, even if the actual memory footprint is reduced due to reduced interference from operating system execution.

As future work, it would be interesting to confirm this hypothesis by monitoring TLB activity through advanced hardware performance counters or through a processor simulator.

5.5.2 Latency

FlexSC and FlexSC-Threads were built primarily with the goal of improving execution efficiency of server class applications and the operating system kernel. For this reason, our evaluation focused on processor metrics as well as overall client observed throughput given increasing loads placed on the server.

A non-intuitive result we showed in this chapter pertains to the effect on the latency of individual client requests. For the application servers we evaluated, servicing each client request at the server side requires the completion of a series of system calls. Specifically in the context of single core experiments, where FlexSC-Threads introduced batching of a large number of system calls, there is potential for increasing the latency of individual system calls, and consequently, latency of requests. However, the results clearly shows a reduction in the observed latency of individual requests.

The reason for this reduced latency also stems from improved efficiency of execution. In the cases of highly loaded server applications, such as the ones we evaluated, the latency of individual *blocking* system call requests does not increase with FlexSC, but actually decreases. With traditional system calls, when an event must block execution of system call (e.g., due to I/O operation) the currently scheduled thread is preempted in favor of a ready thread. In this case, the latency of the blocking system call will involve not only the time for I/O, but the scheduler will also schedule other threads before the original thread is rescheduled. In addition, when threads execute, their non-blocking system calls are completed normally without requiring preemption. The added time to execute these calls add to the latency of the orignal thread blocking call, which must wait for all other threads to execute and be preempted before it has a chance to be rescheduled.

In the case of FlexSC-Threads, with a highly loaded server, the latency of blocking system calls is shorter since both the processing of I/O operations in the kernel is faster and the wait time due to execution of other applications threads. Since both application and kernel execution exhibits improved efficiency, the round trip time involved for these system calls should also improve in relation with the efficiency improvements.

5.6 Summary

In this chapter, we explored a thread based solution to exploit the exception-less system call mechanism presented in Chapter 4. In particular, we presented FlexSC-Threads, a M-on-N threading package that is binary compatible with NPTL and that transparently transforms synchronous sys-

tem calls into exception-less ones. We described how FlexSC-Threads uses the underlying FlexSC system, extracting independent operating system work from the application by relying on the ability to multiplex application threads in user space. Futhermore, we focused on multi-processor concerns in interacting with the kernel FlexSC implementaton and described optimizations that allow for per core data structures, reducing the need for communicating with sibling cores.

With FlexSC-Threads, we demonstrated how FlexSC improves the throughput of Apache by up to 116%, MySQL by up to 40% and BIND by up to 79% while requiring no modifications to the applications. We believe these two workloads are representative of other highly threaded server workloads that would benefit from FlexSC.

In the current implementation of FlexSC, syscall threads process system call requests in no specific order, opportunistically issuing calls as they are posted on syscall pages. The asynchronous execution model, however, would allow for different selection algorithms. For example, syscall threads could sort the requests to consecutively execute requests of the same type, potentially yielding greater locality of execution. Also, system calls that perform I/O could be prioritized so as to issue them as early as possible. Finally, if a large number of cores are available, cores could be dedicated to specific system call types to promote further locality gains.

Chapter 6

Event-Driven Exception-Less Programming

Event-driven architectures are currently a popular design choice for scalable, high-performance server applications. For this reason, operating systems have invested in efficiently supporting non-blocking and asynchronous I/O, as well as scalable event-based notification systems.

We propose the use of *exception-less system calls* as the main operating system mechanism to construct high-performance event-driven server applications. Exception-less system calls have four main advantages over traditional operating system support for event-driven programs: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor based, (2) support in the operating system kernel is non-intrusive as code changes are not required for each system call, (3) processor efficiency is increased since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multicore execution for event-driven programs is easier, given that a single user-mode execution context can generate enough requests to keep multiple processors/cores busy with kernel execution.

We present *libflexsc*, an asynchronous system call and notification library suitable for building event-driven applications. Libflexsc makes use of exception-less system calls through our Linux kernel implementation, FlexSC. We describe the port of two popular event-driven servers, *memcached* and *nginx*, to libflexsc. We show that exception-less system calls increase the throughput of memcached by up to 35% and nginx by up to 120% as a result of improved processor efficiency.

6.1 Introduction

In the previous chapter we described a mechanism that, through no changes in application code or binaries, could leverage exception-less system calls to improve performance of multi-threaded server applications. Hiding the asynchronous nature of exception-less system calls came at a cost: the reliance on a new user-level threading library and the need for a user-level context switch per system call. In this chapter, we explore an alternative approach to using exception-less system

calls in server applications, namely having the program directly use the exception-less system call interface. This approach, however, influences the program structure to extract parallel work to be done by the operating system. Fortunately, a program architecture already exists and has been widely adopted for handling asynchronous requests and events: *event-driven architectures*.

Event-driven application server architectures handle concurrent requests by using just a single thread (or one thread per core) so as to reduce application-level context switching and the memory footprint that many threads otherwise require. They make use of non-blocking or asynchronous system calls to support the concurrent handling of requests. The belief that event-driven architectures have superior performance characteristics is why this architecture has been widely adopted for developing high-performant and scalable servers [75, 144, 146, 161, 198]. Widely used application servers with event-driven architectures include *memcached* and *nginx*.

The design and implementation of operating system support for asynchronous operations, along with event-based notification interfaces to support event-driven architectures, has been an active area of both research and development [17, 36, 31, 69, 82, 110, 113, 144, 146, 198]. Most of the proposals have a few common characteristics. First, the interfaces exposed to user-mode are based on file descriptors (with the exception of *kqueue* [17, 113] and LAIO [69]). Consequently, resources that are not encapsulated as descriptors (e.g., memory) are not supported. Second, their implementation typically involved significant restructuring of kernel code paths into an asynchronous state-machine in order to avoid blocking the user execution context. Third, and most relevant to our work, while the system calls used to request operating system services are designed not to block execution, applications still issue system calls *synchronously*, raising a processor exception, and switching execution domains, for every request, status check, or notification of completion.

In this chapter, we demonstrate that the exception-less system call mechanism is well suited for the construction of event-based servers and that the exception-less mechanism presents several advantages over previous event-based systems:

1. **General purpose.** Exception-less system call is a general mechanism that supports any system call and is not necessarily tied to operations with file descriptors. For this reason, exception-less system calls provide asynchronous operation on any operating system managed resource.
2. **Non-intrusive kernel implementation.** Exception-less system calls are implemented using light-weight kernel threads that can block without affecting user-mode execution. For this reason, kernel code paths do not need to be restructured as asynchronous state-machines; in fact, no changes are necessary to the code of standard system calls.
3. **Efficient user and kernel mode execution.** One of the most significant advantages of exception-less system calls is its ability to decouple system call invocation from execution. Invocation of system calls can be done entirely in user-mode, allowing for truly asynchronous execution of user code. As we show in this chapter, this enables significant performance improvements over the most efficient non-blocking interface on Linux.

4. **Simpler multi-processing.** With traditional system calls, the only mechanism available for applications to exploit multiple processors (cores) is to use an operating system visible execution context, be it a thread or a process. With exception-less system calls, however, operating system work can be issued and distributed to multiple remote cores. As an example, in our implementation of memcached, a single memcached thread was sufficient to generate work to fully utilize 4 cores.

Server (workload)	Syscalls per Request	User Instructions per Syscall	User CPI	Kernel Instructions per Syscall	Kernel CPI
Memcached (memslap)	2	3750	1.25	5420	1.69
nginx (ApacheBench)	12	1460	2.17	6540	2.04

Table 6.1: Statistics about two popular event-driven servers, *memcached* and *nginx*, when running on Linux and using the `epoll` interface. The average number of instructions executed on different workloads before issuing a syscall, the average number of system calls required to satisfy a single request, and the resulting processor efficiency, shown as cycles per instruction (CPI) of both user and kernel execution.

To motivate the use of exception-less system calls for event-driven applications, we measured key execution metrics of two popular event-driven servers: *memcached* and *nginx*. Table 6.1 shows the number of instructions executed in user and kernel mode, on average, before changing mode, for these two servers (Sections 6.3 and 6.4 explain the servers and workloads in more detail.) These applications use non-blocking I/O, along with the Linux `epoll` facility for event notification. Despite the fact that the `epoll` facility is considered the most scalable approach to I/O concurrency on Linux, management of both I/O requests and events is inherently split between the operating system kernel and the application. This fundamental property of event notification systems imply that there is a need for continuous communication between the application and the operating system kernel. In the case of nginx, for example, we observe that communication with the kernel occurs, on average, every 1470 instructions.

We argue that the high frequency of mode switching in these servers, which is inherent to current event-based facilities, is largely responsible for the low efficiency of user and kernel execution, as quantified by the cycles per instruction (CPI) metric in Table 6.1. In particular, as shown in Section 4.2, the frequency of system calls exhibited by both event-driven servers profiled falls within a range where we observe 20% to 70% performance degradation.

Beyond the potential to improve server performance, we believe exception-less system calls is an appealing mechanism for event-driven programming, as: (1) it is as simple as asynchronous I/O to program to (no retry logic is necessary, unlike non-blocking I/O), and (2) more generic than asynchronous I/O, which mostly supports descriptor based operations and which are only partially supported on some operating systems due to their implementation complexity (e.g., Linux does not offer an asynchronous version of the zero-copy `sendfile()`).

One of the proposals that is closest to achieving the goals of event-driven programming with

Mechanism	Invocation	Execution	Retry Logic
Synchronous system call	exception-based	synchronous, blocks on busy resources	No
Non-blocking system call (e.g., `read()` of NONBLOCK fd)	exception-based	synchronous, but never blocks on busy resources (returns error is busy)	Yes
Asynchronous system call (e.g., LAIO or `aio_read()`)	exception-based	partially synchronous and partially async. depending on resource availability	No
Exception-less system call	memory write, no exception	always asynchronous	No

Table 6.2: Comparison of invocation and execution models of different mechanisms used to communicate with the OS kernel.

exception-less system calls is *lazy asynchronous I/O* (LAIO), proposed by Elmeleegy et al. [69]. However, in their system, which is built on an implementation of Scheduler Activations [7], system calls are still issued synchronously, using traditional exception based calls. Furthermore, a completion notification is also needed whenever an operation blocks, which generates another interruption in user execution. For comparison, Table 6.2 summarizes the invocation and execution models of different mechanisms used for communicating with operating system kernels.

6.2 *Libflexsc*: Asynchronous system call and notification library

To allow event-driven applications to interface with exception-less system calls, we have designed and implemented a simple asynchronous system call notification library, **libflexsc**. Libflexsc provides an event loop for the program, which must register system call requests, along with callback functions. The main event loop on libflexsc invokes the corresponding program provided callback when the system call has completed.

The event loop and callback handling in libflexsc was inspired by the *libevent* asynchronous event notification library [151]. The main difference between these two libraries is that *libevent* is designed to monitor low-level events, such as changes in the availability of input or output, and operates at the file descriptor level. The application is notified of the availability, but its intended operation is still not guaranteed to succeed. For example, a socket may contain available data to be read, but if the application requires more data than is available, it must restate interest in the event to try again. With libflexsc, on the other hand, events correspond to the completion of a previously issued exception-less system call. With this model, which is closer to that of asynchronous I/O, it is less likely that applications need to include cumbersome logic to retry incomplete or failed operations.

Contrary to common implementations of asynchronous I/O, FlexSC does not provide a signal or interrupt based completion notification. Completion notification is a mechanism for the kernel to notify a user thread that a previously issued asynchronous request has completed. It is often implemented through a signal or other upcall mechanism. The main reason FlexSC does not offer completion notification is that signals and upcalls entail the same processor performance problems of system calls: direct and indirect processor pollution due to switching between kernel and user

```
1  conn master;
2
3  int main(void)
4  {
5     /* init library and register with kernel */
6     flexsc_init();
7
8     /* not performance critical,
9                     do synchronously */
10    master.fd = bind_and_listen(PORT_NUMBER);
11
12    /* prepare accept */
13    master.event->handler = conn_accepted;
14    flexsc_accept(&master.event, master.fd,
15                  NULL, 0);
16
17    /* jump to event loop */
18    return flexsc_main_loop();
19 }
20
21 /* Called when accept() returns */
22 void conn_accepted(conn *c)
23 {
24    conn *new_conn = alloc_new_conn();
25
26    /* get the return value of the accept() */
27    new_conn->fd = c->event->ret;
28    new_conn->event->handler = data_read;
29
30    /* issue another accept on the master socket */
31    flexsc_accept(&c->event, c->fd, NULL, 0);
32
33    if (new_conn->fd != -1)
34       flexsc_read(&new_conn->event, new_conn->fd,
35                   new_conn->buf, new_conn->size);
36 }
```

```
36 void data_read(conn *c)
37 {
38    char *reply_file;
39
40    /* read of 0 means connection closed */
41    if (c->event->ret == 0) {
42        flexsc_close(NULL, c->fd);
43        return;
44    }
45
46    reply_file = parse_request(c->buf, c->event->ret);
47
48    if (reply_file) {
49       c->event->handler = file_opened;
50       flexsc_open(&c->event, c->fd, reply_file,
51                   O_RDONLY);
52    }
53 }
54
55 void file_opened(conn *c)
56 {
57     int file_fd;
58
59     file_fd = c->event->ret;
60     c->event->handler = file_sent;
61     /* issue asynchronous sendfile */
62     flexsc_sendfile(&c->event, c->fd, file_fd,
63                     NULL, file_len);
64 }
65
66 void file_sent(conn *c)
67 {
68     /* no callback necessary to handle close */
69     flexsc_close(NULL, c->fd);
70 }
```

Figure 6.1: Example of network server using libflexsc. The expected program flow of this example is: (1) `main`, (2) `conn_accepted`, (3) `data_read`, (4) `file_opened`, and (5) `file_sent`. Libflexsc is used to issue system calls asynchronously, and when the system calls are completed, the registered callback is triggered, allowing the execution of the corresponding request to progress into its next stage.

execution.

To overcome the lack of completion notifications, the libflexsc event main loop must poll the *syscall pages* currently in use for completion of system calls. To minimize overhead, the polling for system call completion is performed only when all currently pending callback handlers have completed. Given enough work (e.g., handling many connections concurrently), polling should happen infrequently. In the case that all callback handlers have executed, and no new system call has completed, libflexsc falls back on calling `flexsc_wait()` (described in Section 4.4).

6.2.1 Example server

A simplified implementation of a network server using libflexsc is shown in Figure 6.1. The program logic is divided into states which are driven by the completion of a previously issued system call. The system calls used in this example that are prefixed with "`flexsc_`" are issued using

the exception-less interface (`accept`, `read`, `open`, `sendfile`, `close`). When the library detects the completion of a system call, its corresponding callback handler is invoked, effectively driving the next stage of the state machine. During normal operation, the execution flow of this example would progress in the following order: (1) `main`, (2) `conn_accepted`, (3) `data_read`, (4) `file_opened`, and (5) `file_sent`. As mentioned, file and network descriptors do not need to be marked as non-blocking.

It is worth noting that stages may generate several system call requests. For example, the `conn_accepted()` function not only issues a read on the newly accepted connection, it also issues another accept system call on the master listening socket in order to pipeline further incoming requests. In addition, for improved efficiency, the server may choose to issue *multiple* accepts concurrently (not shown in this example). This would allow the operating system to accept multiple connections without having to first execute user code, as is the case with traditional event-based systems, thus reducing the number of mode switches for each new connection.

Finally, not all system calls must provide a callback, as a notification may not be of interest to the programmer. For example, in the `file_sent` function listed in the simplified server code, the request to close the file does not provide a callback handler. This may be useful if the completion of a system call does not drive an additional state in the program and the return code of the system call is not of interest.

6.2.2 Cancellation support

A new feature we had to add to FlexSC in order to support event-based applications is the ability to cancel submitted system calls. Cancellation of in-progress system calls may be necessary in certain cases. For example, servers typically implement a *timeout* feature for reading requests on connections. With non-blocking system calls, reads are implemented by waiting for a notification that the socket has become readable. If the event does not occur within the timeout grace period, the connection is closed. With exception-less system calls, the read request is issued before the server knows if or when new data will arrive (e.g., the `conn_accepted` function in Figure 6.1). To properly implement a timeout, the application must cancel pending reads if the grace period has passed.

To implement cancellation in FlexSC, we introduced a new *cancel* status value to be used in the status field of the syscall entry (Figure 4.6). When syscall threads in the kernel check for new submitted work, they now also check for entries in *cancel* state. To cancel the in-progress operation, we first identify the syscall thread that is executing the request that corresponds to the canceled entry. This is easily accomplished since each core has a map of syscall entries to syscall threads for all in-progress system calls. Once identified, a signal is sent to the appropriate syscall thread to interrupt its execution. In the Linux kernel, signal delivery that occurs during system call execution interrupts the system call even if the execution context is asleep (e.g., waiting for I/O). When the syscall thread wakes up, it sets the return value to `EINTR` and marks the entry as *done* in the cor-

Server	Total lines of code	Lines of code modifiied	Files modified
memcached	8356	293	3
nginx	82819	255	16

Table 6.3: Statistics regarding the code size and modifications needed to port applications to libflexsc, measured in lines of code and number of files.

responding syscall entry, after which the user-mode process knows that the system call has been canceled and the syscall entry can be reused.

Due to its asynchronous implementation, cancellation requests are not guaranteed to succeed. The window of time between when the application modifies the status field and when the syscall thread is notified of cancellation may be sufficiently long for the system call to complete (successfully). The application must check the system call return code to disambiguate between successfully completed calls and canceled ones. This behavior is analogous to cancellation support of asynchronous I/O implemented by several UNIX systems (e.g., `aio_cancel`).

6.3 Exception-Less Memcached and nginx

This section describes the process of porting two popular event-based servers to use exception-less system calls. In both cases, the applications were modified to conform to the libflexsc interface. However, we strived to maintain the structure of code as similar to the original as possible, to make performance comparisons meaningful.

To reduce the complexity of porting these applications to exception-less system calls, we exploited the fact that FlexSC allows exception-less system calls to co-exist with synchronous ones in the same process. Consequently, we have not modified *all* system calls to use exception-less versions. We focused on the system calls that were issued in the code paths that are involved in handling requests (which correspond to the *hot paths* during normal operation).

6.3.1 Memcached - Memory Object Cache

Memcached is a distributed memory object caching system, built as an in-memory key-value store [75]. It is typically used to cache results from slower services such as databases and web servers. It is currently used by several popular web sites as a way to improve the performance and scalability of their web services. We used version 1.4.5 as a basis for our port.

To achieve good I/O performance, memcached was built as an event-based server. It uses *libevent* to make use of non-blocking execution available on modern operating system kernels. For this reason, porting memcached to use exception-less system calls through libflexsc was the simpler of the two ports. Table 6.3 lists the number of lines of code and the number of files that were modified. For memcached, the majority of the changes were done in a single file (`memcached.c`), and the changes were mostly centered around modifying system calls, as well as calls to *libevent*.

The developers of *memcached* introduced support for multiple processors to memcached, despite most of the native code assuming single-threaded execution. To support multiple processors, memcached spawns worker threads which communicate via a pipe to a master thread. The master thread is responsible for accepting incoming connections and handing them to the worker threads.

6.3.2 nginx Web Server

Nginx is an open-source HTTP web server considered to be light-weight and high-performant; it is currently one of the most widely deployed open-source web servers [161]. Nginx implements I/O concurrency by natively using non-blocking and asynchronous operations available in the operating system kernel. On Linux, nginx uses the `epoll` notification system. We based our port on the 0.9.2 development version of nginx.

Despite having had to change a similar number of lines as with memcached, the port to nginx was more involved, evidenced by the number of files changed (Table 6.3). This was mainly due to the fact that nginx's core code is significantly larger than that of memcached's (about 10x), and its state machine logic is more complex.

We substituted all system calls that could potentially be invoked while handling client requests to use the corresponding version in libflexsc. The system calls that were associated with a file descriptor based event handler (such as `accept`, `read` and `write`) were straightforward to implement, as these were already programmed as separate stages in the code. However, the system calls that were previously invoked synchronously (e.g., `open`, `fstat`, and `getdents`) needed more work. In most cases, we needed to split a single stage of the state machine into two or more stages to allow asynchronous execution of these system calls. In a few cases, such as `setsockopt` and `close`, we executed the calls asynchronously, without a callback notification, which did not required a new stage in the flow of the program.

Finally, for system calls that not only return a status value, but also fill in a user supplied memory pointer with a data structure, we had to ensure that this memory was correctly managed and passed to the newly created event handler. This requirement prevented the use of stack allocated data structures for exception-less system calls (e.g., programs typically use stack allocated "`struct stat`" data structure to pass to the `fstat` system call).

6.4 Experimental Evaluation

In this section, we evaluate the performance of exception-less system call support for event-driven servers. We present experimental results of the two previously discussed event-driven servers: memcached and nginx.

For the results in this chapter, we used Linux kernel version 2.6.33 with our exception-less system call extension (FlexSC). The baseline measurements were collected using unmodified Linux (same version), with the servers configured to use the `epoll` interface. In the graphs shown, we

identify the baseline configuration as "**epoll**", and the system with exception-less system calls as "**flexsc**".

Similar to the experiments in the previous chapter, the experiments presented in this section were run on an Intel Nehalem (Core i7) processor with the characteristics shown in Table 5.1. The processor has 4 cores, each with 2 hyper-threads. We disabled the hyper-threads, as well as the "TurboBoost" feature, for all our experiments to more easily analyze the measurements obtained.

For the experiments involving both servers, requests were generated by a remote client connected to our test machine through a 1 Gbps network, using a dedicated router. The client machine contained a dual core Core2 processor, running the same Linux installation as the test machine.

All values reported in our evaluation represent the average of 5 separate runs.

6.4.1 Memcached

The workload we used to drive memcached is the *memslap* benchmark that is distributed with the libmemcached client library. The benchmark performs a sequence of memcache `get` and `set` operations, using randomly generated keys and data. We configured memslap to issue 10% of set requests and 90% of get requests.

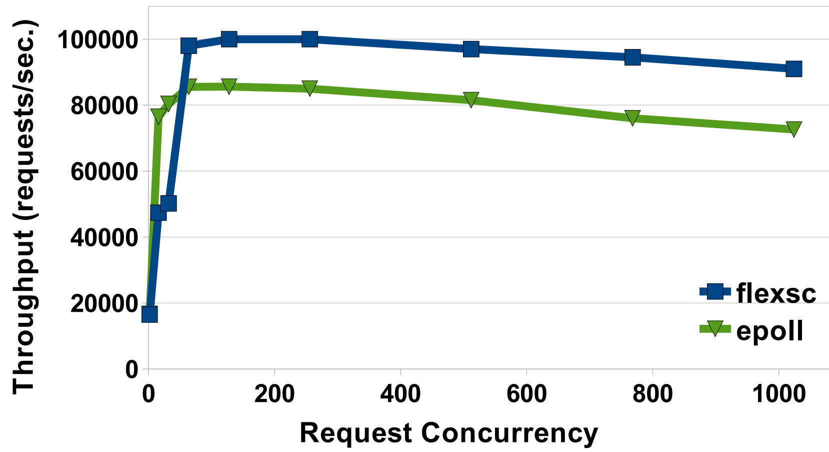
For the baseline experiments (Linux `epoll`), we configured memcached to run with the same number of threads as processor cores, as we experimentally observed this yielded the best baseline performance. For our exception-less version, a single memcached thread was enough to generate enough kernel work to keep all cores busy.

Figure 6.2 shows the throughput obtained from executing the baseline and exception-less memcached on 1, 2 and 4 cores. We varied the number of concurrent connections generating requests from 1 to 1024. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically distributed system calls to other cores to maximize core locality.

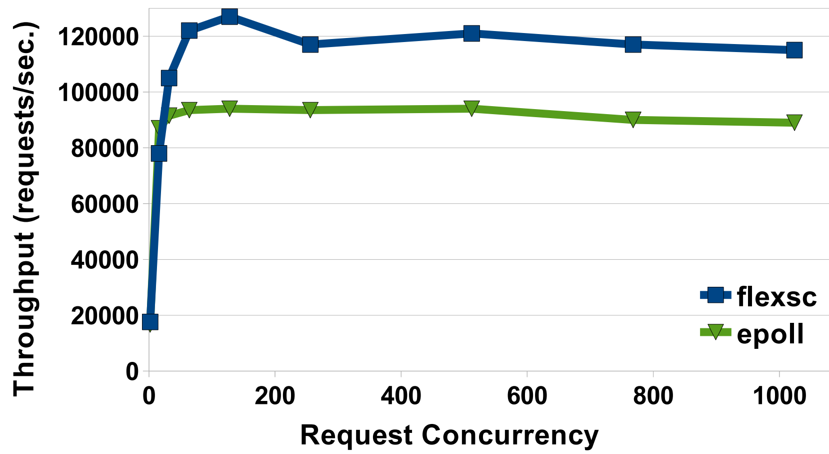
The results show that with 64 or more concurrent requests, memcached programmed to libflexsc outperforms the version using Linux `epoll`. Throughput is improved by as much as 25 to 35%, depending on the number of cores used.

To better understand the source of performance improvement, we collected several performance metrics of the processor using hardware performance counters. Figure 6.3 shows the effects of executing with FlexSC, while servicing 768 concurrent memslap connections (the raw MPKI and CPI values are listed in Table 6.4). The most important metric listed is the cycles per instruction (CPI) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution (the lower the CPI, the more efficient the execution). The other values listed are normalized values of *misses* on the listed structure (the lower the misses, the more efficient the execution).

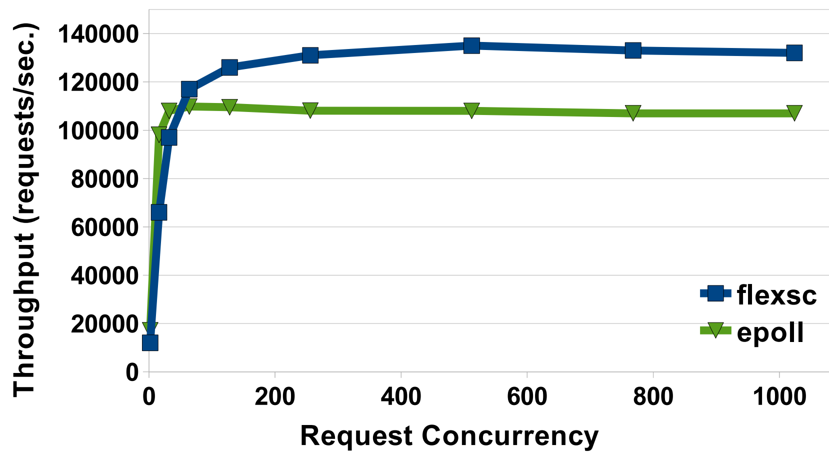
The CPI of both kernel and user execution, on 1 and 4 cores, is improved with FlexSC. On a single core, user-mode CPI decreases by as much as 22%, and on the 4 cores, we observe a 52% decrease in user-mode CPI. The data shows that for memcached the improved execution comes from



(a) 1 Core

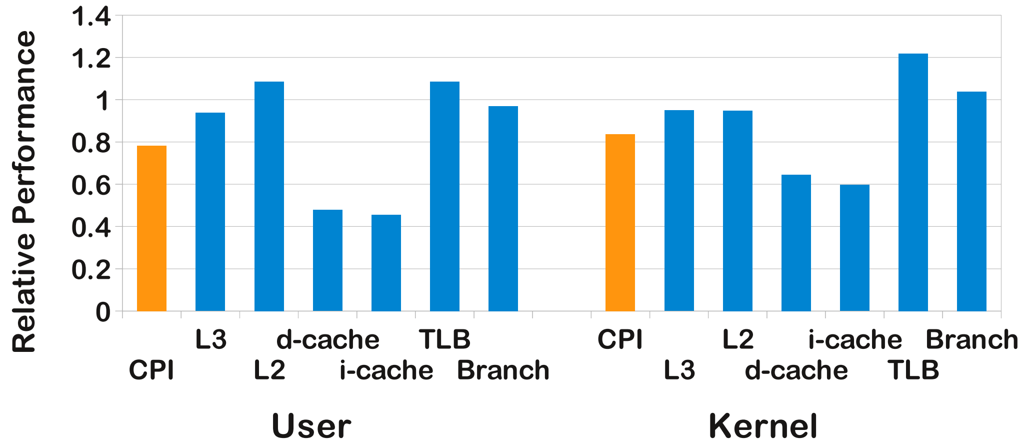


(b) 2 Cores

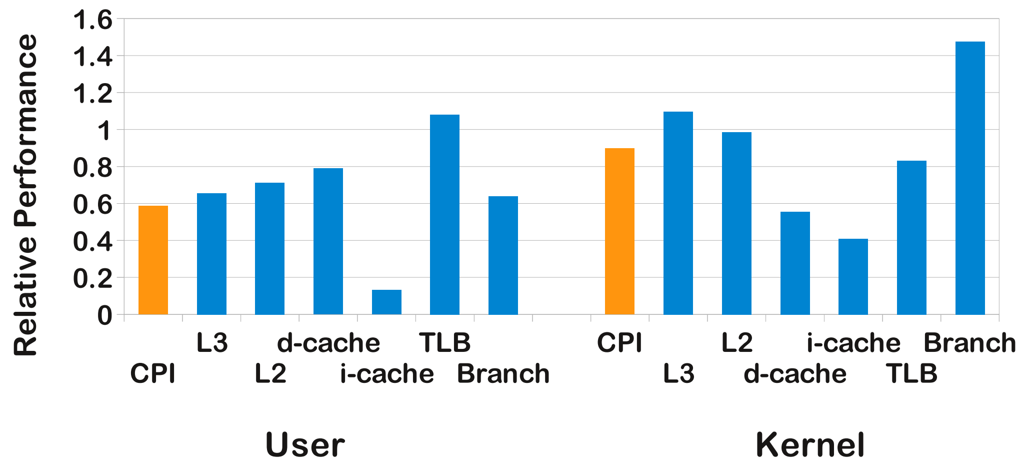


(c) 4 Cores

Figure 6.2: Comparison of Memcached throughput of Linux `epoll` and FlexSC executing on 1, 2 and 4 cores.



(a) 1 Core



(b) 4 Cores

Figure 6.3: Comparison of processor performance metrics of Memcached execution using Linux `epoll` and FlexSC on 1 and 4 cores, while servicing 768 concurrent memslap connections. All values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

Memcached	User							Kernel						
Setup	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch
epoll (1 core)	**1.30**	2.2	8.7	79.3	82.4	9.4	30.8	**1.79**	2.2	21.8	142.3	177.9	5.5	11.7
flexsc (1 core)	**1.02**	2.1	9.4	37.9	37.5	10.1	29.9	**1.49**	2.0	20.7	92.0	106.1	6.7	12.2
epoll (4 cores)	**1.85**	3.9	15.9	77.5	85.9	12.2	31.4	**2.13**	6.1	22.7	105.1	131.1	5.5	11.4
flexsc (4 cores)	**1.10**	2.6	11.3	61.2	11.3	13.0	20.1	**1.92**	6.6	22.4	58.4	53.7	4.6	16.8

Table 6.4: Micro-architectural breakdown of Memcached execution on uni- and quad-core setups. All values shown, except for CPI, are normalized using misses per kilo-instruction (MPKI): therefore, lower values yield more efficient execution and lower CPI.

significant reduction in misses in the performance sensitive L1, both in the data and instruction part (labeled as *d-cache* and *i-cache*).

The main reason for this drastic increase of user CPI on 4 cores is that with traditional system calls, a user-mode thread must occupy each core to make use of it. With FlexSC, however, if a single user-mode thread generates many system requests, they can be distributed and serviced to remote cores. In this experiment, a single memcached thread was able to generate enough requests to occupy the remaining 3 cores. This way, the core executing the memcached core was predominantly filled with state from the memcached process.

6.4.2 nginx

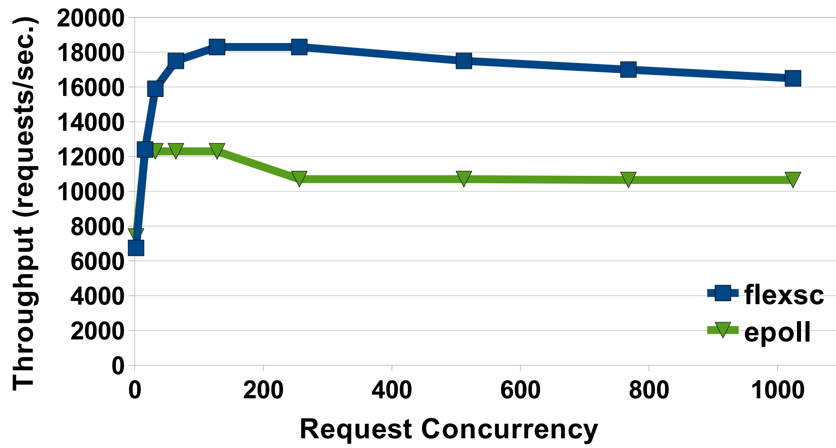
To evaluate the effect of exception-less execution of the nginx web server, we used two workloads: ApacheBench and a modified version of httperf. For both workloads, we present results with nginx execution on 1 and 2 cores. The results obtained with 4 cores were not meaningful as the client machine could not keep up with the server, making the client the bottleneck. For the baseline experiments (Linux `epoll`), we configured nginx to spawn one worker process per core, which nginx automatically assigns and pins to separate cores. With FlexSC, a single nginx worker thread was sufficient to keep all cores busy.

ApacheBench

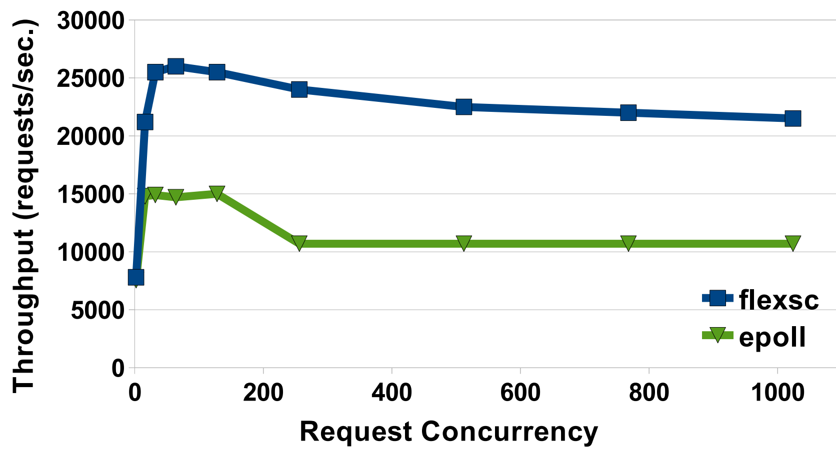
ApacheBench is a HTTP workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 6.4 shows the throughput numbers obtained on 1 and 2 cores when varying the number of concurrent ApacheBench client connections issuing requests to the nginx server. For this workload, system call batching on one core provides significant performance improvements: up to 70% with 256 concurrent requests. In the 2 core execution, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 120% higher throughput, showing the added benefit of dynamic core specialization.

Besides aggregate throughput, latency of individual requests is an important metric when evaluating performance of web servers. Figure 6.5 reports the mean latency, as reported by the client, with 256 concurrent connections. FlexSC reduces latency by 42% in single core execution, and 58% in 2 core execution. It is also interesting to note that adding a second core helps to reduce the average latency of servicing requests with FlexSC, which is not the case when using the `epoll` facility.



(a) 1 Core



(b) 2 Cores

Figure 6.4: Comparison of nginx performance with the ApacheBench when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

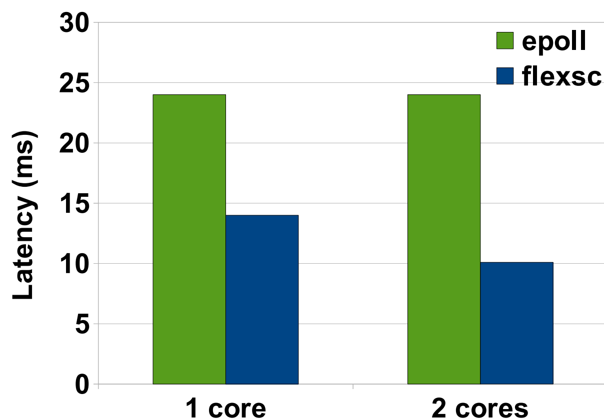
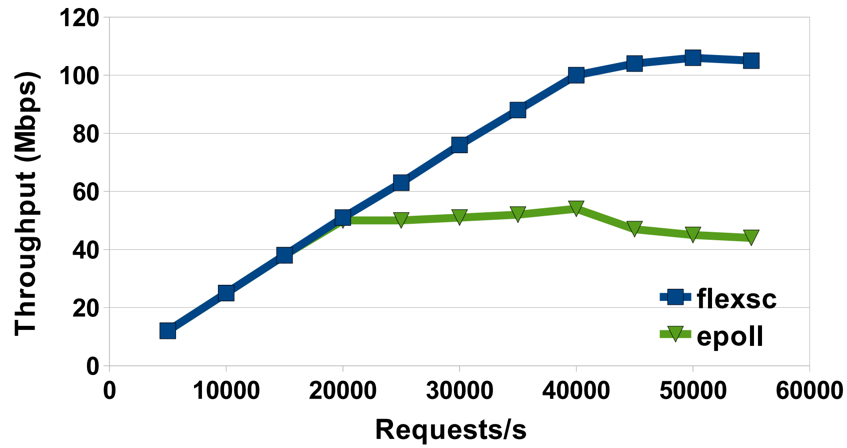
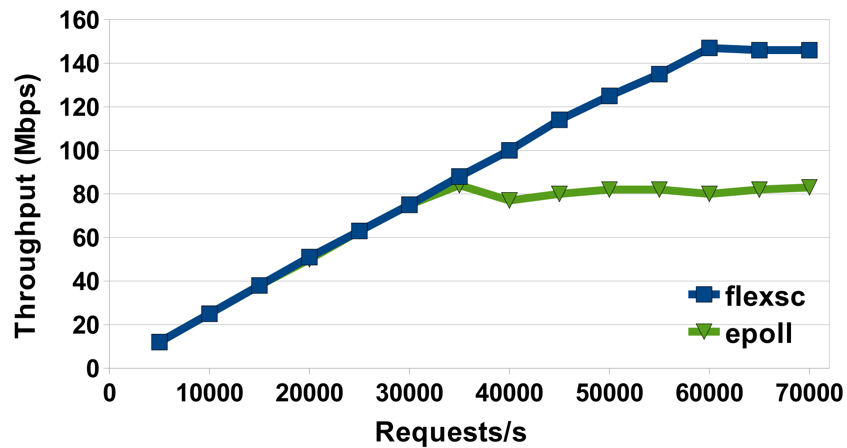


Figure 6.5: Comparison of nginx latency replying to 256 concurrent ApacheBench requests when executing with Linux `epoll` and FlexSC on 1 and 2 cores.



(a) 1 Core



(b) 2 Cores

Figure 6.6: Comparison of nginx performance with the httperf when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

httperf

The *httperf* HTTP workload generator was built as a more realistic measurement tool for web server performance [135]. In particular, it supports session log files, and models a *partially open* system (in contrast to ApacheBench, which models a *closed* system) [170]. For this reason, we do not control the number of concurrent connections to the server, but instead the request arrival rate. The number of concurrent connections is determined by how fast the server can satisfy incoming requests.

We modified httperf (we used the latest version, 0.9.0) in order for it to properly handle large number of concurrent connections. In its original version, httperf uses the `select` system call to manage multiple connections. On Linux, this restricts the number of connections to 1024, which we found insufficient to fully stress the server. We modified httperf to use the `epoll` interface, allowing it to handle several thousand concurrent connections. We verified that the results of our modified httperf were statistically similar to the original httperf, when using less than 1024 concurrent connections.

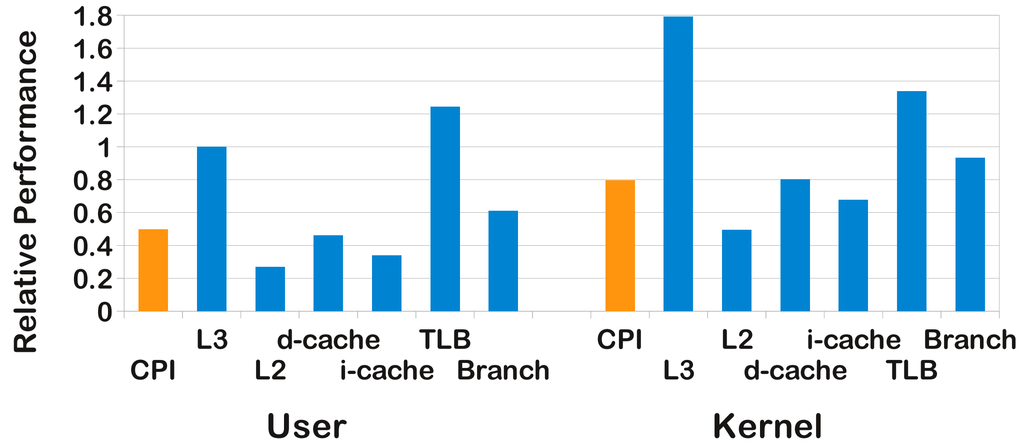
We configured httperf to connect using HTTP 1.1 protocol, and issue 20 requests per connection. The session log contained requests to files ranging from 64 bytes to 8 kilobytes. We did not add larger files to the session as our network infrastructure is modest, at 1Gpbs, and we did not want the network to become a source of bottleneck.

Figure 6.6 shows the throughput of nginx executing on 1 and 2 cores, measured in megabits per second, obtained when varying the request rate of httperf. Both graphs show that the throughput of the server can satisfy the request rate up to a certain value. After that the throughput is relatively stable and constant. For the single core case, the throughput of Linux `epoll` stabilizes after 20,000 requests per second, while with FlexSC, throughput increases up to 40,000 requests. Furthermore, FlexSC outperforms Linux `epoll` by as much as 120% when httperf issues 50,000 requests per second.

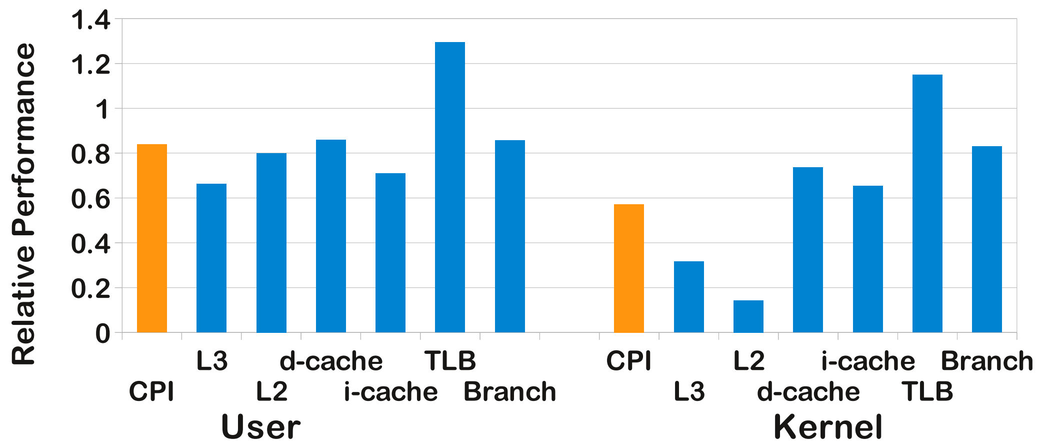
In the case of 2 core execution, nginx with Linux `epoll` reaches peak throughput at 35,000 requests per second, while FlexSC sustains improvements with up to 60,000 requests per second. In this case, the difference in throughput, in megabits per second, is as much as 77%.

Similarly to the analysis of memcached, we collected processor performance metrics using hardware performance counters to analyze the execution of nginx with httperf. Figure 6.7 shows several metrics, normalized to the baseline (Linux `epoll`) execution (the collected MPKI and CPI values are explicitly listed in Table 6.5). The results show that the efficiency of user-mode execution doubles, in the single core case, and improves by 83% on 2 cores. Kernel-mode execution improves efficiency by 25% and 21%, respectively. For nginx, not only are the L1 instruction and data caches better utilized (we observe less than half of the miss ratio in these structures), but the private L2 cache also observes miss rate reduction of less than half of the baseline.

Although we observe increase of some metrics, such as the TLB and kernel-mode L3 misses, the absolute values are small enough that it does not affect performance significantly, as listed in



(a) 1 Core



(b) 2 Cores

Figure 6.7: Comparison of processor performance metrics of nginx execution using `epoll` and FlexSC on 1 and 2 cores, while servicing 40,000 and 60,000 req/s, respectively. Values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

nginx	User							Kernel						
Setup	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch	**CPI**	L3	L2	L1 d$	L1 i$	TLB	Branch
epoll (1 core)	**2.04**	0.0	64.0	105.5	176.6	7.1	25.3	**1.88**	0.5	47.2	101.1	156.6	4.3	12.7
flexsc (1 core)	**1.02**	0.0	17.3	48.8	60.0	8.8	15.5	**1.49**	0.9	23.3	81.0	105.8	5.8	11.9
epoll (2 cores)	**1.92**	0.0	50.3	107.2	168.9	3.7	24.3	**1.92**	3.1	35.0	90.6	138.4	4.8	12.4
flexsc (2 cores)	**1.05**	0.0	21.7	57.0	54.4	8.1	14.7	**1.58**	3.7	23.6	75.2	83.4	4.8	16.2

Table 6.5: Micro-architectural breakdown of nginx execution on uni- and duo-core setups. All values shown, except for CPI, are normalized using misses per kilo-instruction (MPKI): therefore, lower values yield more efficient execution and lower CPI.

Table 6.5. For example, the increase in 80% of kernel-mode L3 misses in the 1 core case corresponds to the misses per kilo instructions increasing from 0.5 to 0.9 (that is, for about every 2,000 instructions, an extra L3 miss is observed). Similarly, the 73% increase in misses of the user-mode TLB in the 2 core execution corresponds to only 4 extra TLB misses for every 1,000 instructions.

6.5 Discussion: Scaling the Number of Concurrent System Calls

One concern not addressed in this work is that of efficiently handling applications that require a large number of concurrent outstanding system calls. Specifically, there are two issues that can hamper scaling with the number of calls: (1) the exception-less system call interface, and (2) the requirement of one syscall thread per outstanding system call. We briefly discuss mechanisms to overcome or alleviate these issues.

The exception-less system call interface, primarily composed of *syscall entries*, requires user and kernel code to perform linear scans of the entries to search for status updates. If the rate of entry modifications does not grow in the same proportion as the total number of entries, the overhead of scanning, normalized per modification, will increase. A concrete example of this is a server servicing a large number of slow or dormant clients, resulting in a large number of connections that are infrequently updated. In this case, requiring linear scans on syscall pages is inefficient.

Instead of using syscall pages, the exception-less system call interface could be modified to implement two shared message queues: an incoming queue, with system calls requests made by the application, and an outgoing queue, composed of system call requests serviced by the kernel. A queue based interface would potentially complicate user-kernel communication, but would avoid the overheads of linear scans across outstanding requests.

Another scalability factor to consider is the requirement of maintaining a syscall thread per outstanding system call. Despite the modest memory footprint of kernel threads and low overhead of switching threads that share address spaces, these costs may become non-negligible with hundreds of thousands or millions of outstanding system calls.

To avoid these costs, applications may still utilize the `epoll` facility, but through the exception-less interface. This solution, however, would only work for resources that are supported by `epoll`. A more comprehensive solution would be to restructure the Linux kernel to support completely non-blocking kernel code paths. Instead of relying on the ability to block the current context of execution, the kernel could enqueue requests for contended resources, while providing a mechanism to continue the execution of enqueued requests when resources become available. With a non-blocking kernel structure, a single syscall thread would be sufficient to service any number of syscall requests.

One last option to mitigate both the interface and threading issues that does not involve changes to FlexSC is to require user-space to throttle the number of outstanding system calls. In our implementation, throttling can be enforced within the libflexsc library by allocating a fixed number of

syscall pages, and delaying new system calls whenever all entries are busy. The main drawback of this solution is that, in certain cases, extra care would be necessary to avoid a standstill situation (lack of forward progress).

6.6 Summary

Event-driven architectures continue to be a popular design option for implementing high-performance and scalable server applications. In this chapter, we proposed the use of *exception-less system calls* as the principal operating system primitive for efficiently supporting I/O concurrency and event-driven execution. We described several advantages of exception-less system calls over traditional support for I/O concurrency and event notification facilities, including: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor-based, (2) support in the operating system kernel is non-intrusive as code changes are not required to each system call, (3) processor efficiency is high since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multicore execution for event-driven programs is easier, given that a single user-mode execution context can generate a sufficient number of requests to keep multiple processors/cores busy with kernel execution.

We described the design and implementation of *libflexsc*, an asynchronous system call and notification library that makes use of our Linux exception-less system call implementation, called FlexSC. We show how libflexsc can be used to support current event-driven servers by porting two popular server applications to the exception-less execution model: memcached and nginx.

The experimental evaluation of libflexsc demonstrates that the proposed exception-less execution model can significantly improve the performance and efficiency of event-driven servers. Specifically, we observed that exception-less execution increases the throughput of memcached by up to 35%, and that of nginx by up to 120%. We show that the improvements, in both cases, are derived from more efficient execution through improved use of processor resources.

Chapter 7

Concluding Remarks

Computing has changed our society dramatically over the last 50 years and is positioned to play a central role in many future human endeavors. The evolution in both the performance of computers and the capacity to manipulate large amounts of data has been key to the drastic increase in the applicability of computers to scientific research, industrial automation, information dissemination, communication and many other activities. It is our belief that the continued investment in improving the performance, accessibility, and cost of computers will likely yield benefits to our society and economy.

Since the birth of computing, and the subsequent widespread adoption of computers, performance has improved at staggering rates through hardware upgrades alone. Due to engineering and physical limits, this trend changed abruptly circa 2005. We can no longer expect doubling of single-threaded application performance from one generation of hardware to the next. Thermal-power issues have influenced processor manufacturers to instead focus on providing an increasing number of cores. Another aspect of modern computers that highly influences their performance is the memory performance gap. The memory performance gap, rooted in the speed differences between processors and large off-chip memory devices (e.g., DRAM), has been widening quickly since the 1980s. As a result, increases in the on-chip cache capacity has been an on-going trend for the past three decades. For these reasons, we believe that *parallelism* and *communication*, whether explicitly through synchronization primitives or implicitly through the cache and memory hierarchy, will continue to play a central role in computer performance.

In this dissertation, we focused on improving performance by reducing implicit communication due the pollution of processor structures. Pollution occurs in processor structures when content that will be reused (accessed in the near future) is replaced in favor of content that will not be reused. In particular, we argue that the run-time and operating system should play a role in helping reduce processor state pollution. In addition, we believe that the run-time and operating system layer is the most natural layer of the computer stack to incorporate certain optimizations, including the ones introduced in this dissertation.

To support our main thesis, we developed two novel operating system mechanisms, demon-

strating how these mechanisms can be used by run-time libraries or directly by applications to reduce pollution in key processor structures. The key contributions of this work are:

- We developed an operating system cache filtering service, that is applied at run-time and improves the effectiveness of secondary processor caches. We identified intra-application interference as an important source of pollution in secondary on-chip caches. Leveraging commodity hardware performance units, we demonstrated how to generate application address space cache profiles at run-time with low overhead. The online profile is used to identify regions of memory or individual pages that cause pollution and do not benefit from caching. Finally, we showed how page-coloring can be used to create a *software pollute buffer* in secondary caches to restrict the interference caused by the polluting regions of memory.

- We developed a novel mechanism, called exception-less system call, that allows applications to request operating system services with low overhead and asynchronously schedule operating system work on multiple cores. We quantified the impact of traditional exception-based system calls on the performance of system intensive workloads, showing that there are direct and indirect components to the overhead. We proposed a new system call mechanism, exception-less system calls, that uses asynchronous communication through the memory hierarchy. An implementation of exception-less system calls, called FlexSC, is described within a commodity monolithic kernel (Linux), demonstrating the applicability of the mechanism to legacy kernel architectures.

- We developed a new hybrid threading package, FlexSC-Threads, specifically tailored for use with exception-less system calls. The goal of the presented threading package is to translate legacy system calls to exception-less ones *transparently* to the application. We experimentally evaluated the performance advantages of exception-less execution on popular server applications, showing improved utilization of several processor components. In particular, our system improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 79% while requiring no modifications to the applications.

- We explored exposing exception-less system calls directly to applications. To this end, we developed a library that supports the construction of event-driven applications that are tailored to request operating system services asynchronously. We showed how to port existing event-driven applications to use our new mechanism. Finally, we identified various benefits of exception-less system calls over existing operating system support for event-driven programs. We showed how the use of direct use of exception-less system calls can significantly improve the performance of two Internet servers, *memcached* and *nginx*. Our experiments demonstrate throughput improvements in *memcached* of up to 35% and *nginx* of up to 120%. As anticipated, experimental analysis shows that the performance improvements largely stem from increased efficiency in the use of the underlying processor when pollution

is reduced.

In the remainder of this chapter, we describe some of the lessons learned while pursuing the research described in this dissertation. Next, we conclude the dissertation by outlining a few research avenues that either extend or are influenced by the concepts presented in earlier chapters.

7.1 Lessons Learned

Research in software systems is an inherently empirical endeavor. As a consequence, throughout the development of the research presented in this dissertation we iterated through several unsuccessful designs and implementations of our prototypes. This dissertation contains the description of the most mature stage of our work, since it summarizes the elements and lessons of our most successful work and demonstrates the greatest potential to advancing the performance of computer systems. Nonetheless, there are a few lessons that we learned while conducting our research.

7.1.1 Difficulty of assessing and predicting performance

One issue that repeatedly challenged the development of our work was that of assessing and predicting performance. Our understanding of the sources of performance inefficiencies or anomalies was achieved through trial and error. Furthermore, once understood, predicting the outcome of changes to the system proved to be difficult with often unexpected results.

Nonetheless, we have found that detailed characterization and analysis of the different interacting components that influence the performance of a workload to be helpful. In particular, hardware performance counters have allowed us to obtain feedback from the hardware and better understand the predominant symptoms of inefficiencies. Despite the potentially detailed feedback available through hardware performance counters, mapping this information to higher level concepts such as program code or interference between processes, is still a manual and error-prone task. Most existing tools for code profiling using hardware performance counters are simple, leaving the user to interpret the results.

As a concrete example, initial versions of our FlexSC system exhibited poor performance and significant overhead with respect to baseline Linux execution. Profiling execution did show various potential sources of overhead, but not a single major source of inefficiency. Through an iterative process of optimizing our implementation over several weeks, the performance improved significantly. Yet the performance improvement obtained through each iteration of the process was difficult to determine ahead of time. Addressing the sources of overhead that were highest ranked in the performance counter profile proved to yield modest performance improvements. However, midway through the process of optimization we observed increasing performance improvements.

Only in hindsight were we were able to identify how multiple sources of overhead influenced performance. It was often the case that only upon resolving the last source of measurable overhead

did we observe the benefits of removing the previous sources of overhead. In general, we found it difficult to *predict* ahead of time how each individual change would impact the performance of the system as a whole. For this reason, we believe that performance profiling methodologies and tools need to advance in order to provide insightful and accurate information to developers. We provide some suggestions in Section 7.2.6.

7.1.2 Run-time use of hardware performance counters

Hardware performance counters were originally introduced as a mechanism for computer architects to aid in *post-silicon validation* (i.e., to debug and verify processor behavior after fabrication). Soon after, it was found that using these counters were also useful for performance profiling. With feedback from the hardware, application programmers and compilers could target sources of inefficiency more accurately. In our work, we have advocated for the use of hardware feedback *at run-time*. We believe that this type of feedback has potential for many other optimizations within the context of just-in-time compilers, run-time and operating systems.

Unfortunately, current implementations of hardware performance counters are not targeted for run-time use. Issues we have encountered when using hardware performance counters at run-time include: high overhead of accessing and reprogramming them, few physical registers which prevents concurrent counting of several hardware events, inability to add more complex logic than just counting simple events, and lack of precision when correlating events with addresses (instruction or data).

To make the most out of the current performance monitoring units (PMUs), we have adopted two strategies that have made PMUs amenable to run-time use. First, we have found that most of the tools available for accessing the PMU impose high overheads, and yet at the same time allow for only limited flexibility in programming the PMU. Part of the reason for this is the focus on supporting performance profiling done through multiple runs and using offline processing and analysis. As a result, we have found that constructing our own module, included in the operating system kernel, has helped to overcome the shortcomings of existing tools. Building our own module has allowed significant lower overheads since processor interrupts, which is how the PMU typically notifies software of programmed events, can be efficiently handled within the operating system kernel. In addition, customizing PMU behavior (e.g., multiplexing the physical registers, processing results from counters, keeping a log of addresses or events, etc.), can be done more efficiently and easily in a custom made kernel module. In our work, we benefited from and extended infrastructure initially developed by Azimi et al., who identified performance advantages of an in-kernel PMU module for multiplexing of events [13].

The second strategy we have used to make current hardware performance counters amenable to run-time is relying on statistical methods. We advocate the use of techniques that can infer characteristics from sampled data points and work well with imprecise and/or noisy information. There are two main advantages in relying on statistical methods that can extract information from

sampled data points: lower overhead and ability to cope with imprecise information. Given the current high overhead of collecting samples at high frequency (e.g., on every memory instruction), statistical based techniques can trade-off precision for lower overhead by reducing the frequency of data collection. In addition, we have observed in practice that inherent design issues of hardware performance counters impact the precision of data samples. As a consequence, it is important to be able to cope with data that is occasionally imprecise.

An example of this type of technique is the *statistical multiplexing* of performance counters developed by Azimi et al. [13]. It allows developers to overcome the limited the number of physical registers available in PMUs by fine-grain multiplexing of logic counters onto the physical counters. They use a simple interpolation function to infer the missing values, and experimentally verify that the interpolated values are statistically close to the real values.

The ROCS system described in Chapter 3 also uses sampling in two ways. First, to build page-level cache profiles, as described in Section 3.3.1, only a small proportion of cache accesses are monitored and sufficient to create a useful profile. Second, continuously profiling for the entire duration of the application would introduce prohibitively high overhead. Therefore we built a cache profile per phase of the application by monitoring a short slice of execution. This method provided sufficient accuracy to classify the pollution behavior of pages throughout each phase of the application.

7.1.3 Interference of prefetching on caching

While performing experimental work related to Chapter 3, we were surprised to observe the influence that prefetching can have on the caching behavior of data items. As a concrete example, we observed a region of memory that, with prefetching disabled, exhibited an LRU friendly access pattern. Consequently, we assumed that the region of memory should be given space in secondary caches to allow for data reuse. However, our experiments when run with prefetching enabled showed that the cache worked best when that region of memory was placed in the pollute buffer (i.e., was given a small portion of the cache). The reason for this unexpected result was that the region of memory was not only accessed in an LRU friendly way, but also in a way that allowed for the prefetcher to be highly effective. Given the effectiveness of prefetching, it turned out that placing the data in the cache was not reducing the number of misses to that region of memory. Instead this region of memory wasted space in the cache, and subsequently harmed application performance.

Another example of prefetching interference is documented in Section 3.3.3 and visible in Figure 3.9. In the figure, we see that some of the memory regions exhibit similar cache behavior with and without prefetching (virtual pages 0 to 20,000), and some memory regions observe significant changes to both the number of accesses in L2, as well as their miss rates (virtual pages greater than 20,000).

This interference may be easy to understand in hindsight, however, we found there are few

caching studies in the literature that account for this interference. In particular, several studies were presented recently aiming to improve the performance of last-level caches, yet these studies do not consider the impact of prefetching [44, 95, 154, 155]. Current trends such as increasing sizes of secondary caches, more aggressive prefetchers, as well as caches that are shared between multiple cores on a chip, will likely exacerbate the interference between caching and prefetching.

Since all mainstream processors include data prefetchers (sometimes, multiple prefetchers are included), we advise against studying caching without considering the impact of prefetching. Such studies may yield insights that are difficult to transfer to real and upcoming systems.

7.1.4 Cost of synchronization

The difficulties of constructing scalable software systems have been extensively explored in both academia and industry (refer to Section 2.2.2 for more information). In particular, the Tornado operating system, which was developed in the mid 1990s, was among the first to show the importance of locality and independent execution to scalability of operating systems [83]. Despite these past lessons, we feel it is worth reiterating the importance of locality and independence of execution to the performance of parallel software.

During the development of both our FlexSC and FlexSC-Threads prototypes — specifically when tuning the performance for multiple processors — we observed high overheads due to communication between cores and synchronization. In fact, even modest amounts of synchronization observably deteriorated performance. Part of the reason relates to the implementation of current atomic primitives in the processors we explored (Intel x86-64 and PowerPC 64). We have observed that introducing one or two atomic operations is sufficient to impact performance *even when accesses are uncontended*. With the promise of increases in the number of cores per chip, the cost of atomic operations is unlikely to decrease in the future, particularly considering that mainstream multicore designs apply memory coherence to all cores.

The communication and synchronization costs have motivated the heavy use of per core data structures, run queues and syscall pages in FlexSC and FlexSC-Threads, as described in Sections 4.4.4 and 5.2. The per core design of data structures along with per core threads mirrors some of the design principles behind Tornado. With per core structures, the use of atomic operations is minimized to specific situations that require coordination with multiple cores. In addition, data is more likely to be placed effectively in the cache hierarchy since we expect reduced number of coherence invalidations and reduced number of redundant copies in multiple caches.

7.2 Future Work

In this section we outline a few avenues of research that extend or are closely related to the work presented in this dissertation.

7.2.1 *Hardware Introspection* through advanced hardware performance counters

In Chapter 3, we presented an operating system technique that dynamically categorized the pollution behavior of application pages. To do so, we relied on information provided by hardware performance counters. Using alternative approaches such as dynamic binary instrumentation or emulated execution would have been impractical due to unacceptably high overheads. We believe that the ability for hardware to introspectively provide information about execution and its effect on different hardware components can open opportunities to understand and adapt to performance issues at run-time.

Unfortunately, current support for hardware introspection, which is mainly centered around hardware performance counters, debug registers and a couple of bits in the page-table, is lacking in several aspects. Shortcomings of existing hardware introspection include high overhead, lack of flexibility, inability to precisely monitor certain events (e.g., multiple concurrent memory instructions in the pipeline), lack of documentation and lack of standardization among multiple processor versions or families.

Addressing these shortcomings is largely a responsibility of the processor industry. Nonetheless, advancing potential software uses of hardware introspection should provide incentives for mainstream adoption. We provide a list of features that we believe have several potential uses, and are sufficiently general purpose that they should be considered for mainstream adoption:

- Memory tracing and profiling. A common use of hardware performance counters, such as that in Chapter 3, is to generate traces of memory accesses. In our case, we were interested in a sample of accesses to secondary cache levels of the memory hierarchy. However, other researchers have used memory tracing to support various other run-time techniques such as to just-in-time generate software prefetches [2], determine thread affinity [187], attribute cache misses to dynamically allocated data structures [147], track page-level access information to improve OS memory management [12], and generate miss rate curves for shared cache management [188].

  Unfortunately, implementing software that can efficiently and correctly generate memory traces using hardware performance counters is difficult and time consuming. Overcoming the high overheads associated with frequently accessing the performance counters along with the inability to easily filter unwanted events requires carefully constructed software and customized techniques.

- Interrupt-less profiling by spilling performance data to memory. One solution to the high overhead of collecting memory traces (mentioned above), is to allow samples to be collected without interrupting execution. Instead of requiring registers to be read at interrupt time, information could be buffered in memory and read periodically. This strategy can drastically reduce the overhead of data collection due to frequent mode switches and interrupt handling.

To the best of our knowledge, only the Intel Precise-Event Based Sampling (PEBS) infrastructure allied with the use of the Debug Store Area provides such a mechanism [90]. Unfortunately, the state saved by the Intel mechanism cannot be configured and a total of 176 bytes, which includes all the architectural registers, is saved per sample. Consequently, the overhead of generating samples and subsequently processing the logged information is much higher than necessary.

- Programmable logic. One way that could reduce the overhead of using hardware performance counters is to introduce programmable behavior to count and report hardware events. In the examples above that use memory tracing, one programmable behavior could be filtering unwanted events and/or saving only a subset of context information (e.g., the virtual memory address, but not the register information).

  Another programmable behavior would be to switch between the set of interested events automatically. For example, to monitor inter-core communication during critical sections of programs, the PMU could be configured to monitor a locked instruction or the instruction pointer corresponding to a lock function, and automatically switch to counting inter-core cache line communication. With current PMUs, this is not possible, and counting both events concurrently does not allow us to infer the desired information.

- Standardization. Hardware performance counters are, in their current form, intrinsically tied to the processor architecture and micro-architecture. So far, processor manufacturers that have included hardware performance counters have done so with custom designed interfaces, and custom lists of events that can be monitored. In fact, sometimes multiple revisions of the same family of processors will implement hardware performance counters with incompatible interfaces. This incompatibility, and the lack of a promise to maintain specific counters in the future, poses a burden on software designers who may otherwise benefit from relying on hardware introspection.

  We believe that despite the challenges, there are several architectural and micro-architectural feature that are common to most general purpose processors. These characteristics could easily be standardized, along with the software interface to the PMU. In addition, with the commitment from hardware designers to maintain the common counters in future processor designs, we believe that the software industry would be more open to adopting the use of hardware performance counters.

Furthermore, other researchers are also exploring uses of hardware instrospection for different purposes. Log-based architectures advocates hardware extensions to reduce the overheads of run-time monitoring, specifically targetting multi-processor uses [46, 47]. Other researchers have used hardware based run-time monitoring for allowing developers to better understand how data is accessed in complex software [139, 189].

7.2.2 Hardware support for *event-based code injection*

A hardware feature that we believe would open a number of opportunities for dynamically adapting applications, run-time and operating systems is *event-based code injection*. Event-based code injection would work similar to microcode assist, which is a technique widely used by processor manufacturers to implement (or correct, post-silicon) complex features or instructions. In essence, a series of microcode instructions are placed by the firmware in on-chip storage and injected into the pipeline when certain pre-programmed events occur or instructions execute (e.g., TLB miss handling, rare floating-point corner case, and hypervisor mode switch).

Our proposed event-based code injection, however, could operate using ISA instructions and not necessarily microcode and would allow applications and run-time systems to reprogram the code to be injected. In addition, instead of relying on a static list of events provided by the firmware, this mechanism should be programmable at run-time. In particular, it would be valuable to couple code injection with the performance monitoring unit, so that the existing PMU events could be used as triggers for programmable code injection.

With hardware support for event-based code injection, achieving some of the goals mentioned in the previous subsection becomes simple. For example, it would be possible to create a trace of cache misses while at the same time spilling the trace to memory. To do so, we could program the PMU to inject code upon cache misses and construct the injected code to store, in a pre-allocated portion of memory, the value of the operand (address) of the instruction that raised the event (i.e., cache miss). In addition, creating a log of TLB misses would be analogous to creating a trace of cache misses, with the only difference being that we would program the PMU to monitor TLB misses. We believe it would be possible for software to implement this type of monitoring with just a few assembly instructions.

Similarly, event-based code injection could be used for run-time analysis of inter processor communication. With enough precision, this mechanism would be useful not only for monitoring for the purposes of performance analysis and optimization, but also for debugging. For example, dynamic race condition detection may be efficiently implemented by detecting coherence invalidations of accessed data items during critical sections. In our proposed system, instrumentation can be injected when the processor detects coherence invalidations or accesses to previously invalidated cache lines.

7.2.3 Exposing *software buffer* to language or compiler

The ROCS mechanism described in Chapter 3 is an operating system technique meant to be used at run-time. At a high level, there are two separate components used in ROCS: a profiling component that identifies cache polluting pages, and a *software pollute buffer* component that restricts cache occupancy for selected pages. The greatest advantage of the profiling component is that it allows ROCS to improve cache utilization without prior knowledge of application's access patterns and

works completely transparent to applications. However, as discussed in Section 3.7.1, the runtime profiling has a few limitations including potentially high overhead and its effectiveness being conditioned on the repetition of accesses to memory.

One potential avenue of research is to explore the use of *software pollute buffers* without the runtime profiling component. The operating system could provide a new allocation primitive that allocates pages mapped to the software pollute buffer partition of the cache. User-space could manage its data structures, moving them to this specially allocated memory when there is an expectation that the data will cause cache pollution.

For example, programmers interested in tuning application performance may be willing to provide hints so that specific data be placed in the software pollute buffer. A simple example for using hints to guide data placement in the software pollute buffer is the case of large data structures with single-use streaming access patterns (sometimes referred to as linear scanning), which are present in programs that perform data compression, simple image processing, text search, among others. A similar strategy would be to augment the compiler to predict sequential and streaming access patterns and allow the compiler to manage memory mapped to the software pollute buffer transparently to the programmer.

7.2.4 Software assisted cache management

In the late 1950s and early 1960s, a research team at the University of Manchester introduced the concept of *virtual memory* in the design of the Atlas computer [77, 107]. Motivated by the desire to allow programmers to easily access more memory than the limited available physical memory (96 KB), they designed a system that allowed applications to access 6 to 7 times more memory than the actual physical capacity, while being completely transparent. And while their initial implementation of the virtual memory hardware extensions and operating system support for paging did not outperform manually managing main memory and backing store, it allowed less experienced programmers to fully utilize the resources of the Atlas computer. By the end of the 1960s, however, other researchers and computer makers had improved the performance of virtual memory hardware along with operating system algorithms so that they started to outperform a manually managed memory hierarchy [168]. Since then, virtual memory has been shown to provide other benefits such as isolation between applications, isolation between privilege levels, and ability to offload memory fragmentation resolution to the operating system.

We believe that the time has come for operating systems to assist in the management of processor caches, specifically the secondary levels of cache. We briefly outlined a few of the reasons behind software assisted cache management in in Section 3.7.3. Overall, the cache analysis of per application address space presented in Section 3.3 provides some evidence that a coarse-grain view of the application may be helpful for cache management. In fact, as processor caches grow in both capacity and levels (e.g., the impending die stacking (3D) technology [32, 108]) the coarse grain view of the computer's memory, including information about virtual machines, operating systems

and applications becomes a valuable asset in managing the cache hierarchy.

Furthermore, the ubiquity of multicore processors has added a new set of challenges in guaranteeing cache isolation and low communication costs between cores or applications [119, 156, 186, 188]. Given that the operating system is the central manager that enforces isolation and priorities of different applications, it is the most suitable layer to manage the cache hierarchy in these regards, as well.

It is not clear what functionality that hardware should expose to the operating systems and what will be required by the interface. However, we believe that this topic should be investigated to enhance our understanding of how software can assist cache management and its overall performance merits. We hope that it will provide incentives for processor manufacturers to incorporate such a hardware extension in future designs.

7.2.5 Lightweight inter-core notification and communication

In current mainstream multiprocessor architectures, whether multicore or multichip, there are two mechanisms available for inter-processor communication. The first way is simply through shared memory, since current architectures implement coherent memory and cache hierarchies. In FlexSC, we showed how coherent memory can be leveraged to build an inter-core communication facility by implementing a simple protocol to determine when messages are ready to be consumed. The second way, is the *inter-processor interrupt* (IPI) which is a synchronous notification mechanism. With an IPI, a sender processor can raise interrupts on a desired set of processors. The handling of IPI typically follows a simple protocol, which checks for pending messages or requests through coherent shared memory.

In our experience of building FlexSC, we found that these two mechanisms were sufficient to build inter-core communication facilities. Yet, there are shortcomings to both facilities that required FlexSC to efficiently deal with them. First, with respect to using coherent memory, we found that its *reactive* nature to be inefficient in certain cases. For example, if a sender core sends a message to a receiver core (say, through an IPI), the receiver core will *pull* cache lines from the senders cache when it attempts to access shared data (therefore invoking the coherence protocol). This reactive nature of coherence has the unfortunate side-effect that the receiver core must *stall* and wait for the lines to be transferred (sometimes, one at a time, which amplifies the observed overhead).

Given that the reactive nature of coherence can negatively impact the performance of communication between cores, we propose that a sender core be allowed to *push*, proactively, cache lines (potentially in bulk) to receiver cores. Such an operation could be done asynchronously, similar to how prefetch instructions are implemented today, not requiring the sender to stall. On the receiver side, by the time it is ready to use the communicated data, it is possible that the lines have been partially or fully transferred to its local cache.

With respect to IPIs, current implementations are surprisingly slow, often taking several thou-

sands of cycles to deliver a single IPI. The reasons for this are partially historic; the IPI mechanism was introduced when the only multi-processors available were multichip processors. As a result, the IPI mechanism was implemented in the context of the I/O controller, since I/O was the only other source of external interrupts.

We believe that with the need to build efficient parallel applications, it is necessary to redesign the inter-core notification mechanism. In our view, there are two unmet requirements in today's IPI. First, as already mentioned, the high overhead of sending and receiving IPIs is unacceptable for various uses. In FlexSC, for example, we decided to use a hybrid strategy that relies mostly on polling of memory (akin to soft timers [11]), and sometimes used IPIs, so that in the common case, we would avoid the IPI overhead.

The second unmet requirement is fast user-level inter-core notification. In the case of user-space notification, the IPI overheads are exacerbated due to the need to involve the operating system on both the sender and receiver. Because of this issue, programmers have developed alternative notification facilities that use coherent memory allied with polling (e.g., the way *spin-locks* poll on a memory location to emulate an explicit notification). These strategies are difficult to make efficient and scalable and can inadvertently complicate software development.

7.2.6 Interference aware profiling

Traditional application performance profiling, whether using hardware performance counters, simulation or code instrumentation, tries to attribute resource consumption to portions of program code. In fact, the most common resource to be monitored is processor cycles or, simply, time. This strategy is clearly effective at providing useful profiling information, allowing programmers to identify inefficient algorithms, data structure implementations, and program hotspots. The DCPI project was one of the first work that attempted to attribute reasons (e.g., cache misses, branch mispredictions) to stalled cycles, giving developers more insight into the performance of their code [6].

However, our work has shown a type of inefficiency that is not easily captured by traditional performance profiling, namely, performance *interference*. We have observed that interference can occur whether within the context of a single application, or between different software components, such as the application and operating system. Interestingly, using traditional methods of profiling to identify inefficiencies in the execution of system intensive workloads did not provide the insights described in Section 4.2, where we describe the costs of synchronous system calls. Instead, traditional profiling displayed what is commonly known as a *flat profile*; a profile in which no single portion of code accrues sufficient samples to either differentiate itself from the remainder of the code or offer significant optimization opportunities. Given a flat profile, it becomes difficult for developers to further optimize the application performance.

We propose to augment traditional profilers with *interference aware profiling*. We envision that interference aware profiling be used to detect contention of resources that occur through either concurrent execution (on multicore systems) or through time sharing of resources. With such a

profile, we believe it would have been easier to detect intra-application interference in secondary caches as well as the operating system interference due to frequent system calls. Showing developers which pieces of software have affinity due to implicit sharing of data and/or instructions, and which are incompatible would allow them to make informed decisions in potentially redesigning their code and data structures. Furthermore, if interference information could be cheaply provided to a run-time system, scheduling of execution can also be made in a way to reduce the detected interference, and leverage existing affinities (e.g., thread clustering [187]).

7.2.7 Execution slicing: pipelining execution on multicores

The experimental evaluation of our FlexSC system showed that clustering execution with high sharing affinity onto specific cores, while separating execution of code that is likely to cause interference, improves the efficiency and overall performance of parallel execution. The resultant *specialized cores* has also been shown to be useful in different contexts [42, 101, 112, 208]. In our work, we leveraged existing boundaries between execution domains (user and kernel) which intuitively should have mostly disjoint data and instruction accesses.

There is a large portion of parallel applications that do not suffer from operating system interference and for which FlexSC would provide no performance benefit. Nonetheless, as shown in Chapter 3, intra-application interference can occur in practice, specially in applications with large working sets and complex access patterns. We stipulate that similar intra-application interference also exists in highly parallel applications, and in particular, given our experience with server type applications, we believe there are naturally occurring portions of code with both high affinity and high interference.

A potential technique for augmenting performance of highly parallel applications is to apply the *interference aware profiling* (described in the previous subsection), with a user-level asynchronous execution platform, similar to the one provided with FlexSC. To do so, we would use the run-time profile to determine portions of execution that exhibit high affinity and *slice* the code in between portions. Concretely, slicing could be implemented by replacing the edges of the control flow graph, through simple binary rewriting, with a call to the run-time system. This way, the run-time can insert scheduling points in between these high affinity slices. Subsequently, execution could be scheduled on separate cores, potentially forming a software pipeline, where execution is migrated from core to core. Ideally, each core would be *specialized* to execution related to a single slice of the program. With enough parallelism, communication between slices could take place mostly asynchronously, in the same spirit as the FlexSC user-kernel communication. We expect that execution slicing allied with multicore scheduling of slices would improve the efficiency of parallel applications by reducing interference and competition in processor structures.

Bibliography

[1] Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *USENIX Summer Technical Conference* (1986), pp. 93–113.

[2] Adl-Tabatabai, A.-R., Hudson, R. L., Serrano, M. J., and Subramoney, S. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2004), PLDI '04, pp. 267–276.

[3] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference* (2002), USENIX ATC'02, pp. 289–302.

[4] Agarwal, A., Hennessy, J., and Horowitz, M. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems 6*, 4 (November 1988), 393–431.

[5] Ahmad, I., Gulati, A., and Mashtizadeh, A. vIC: Interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (2011), USENIX ATC'11, pp. 45–58.

[6] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. Continuous Profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems 15*, 4 (1997), 357–390.

[7] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems 10*, 1 (1992), 53–79.

[8] Appavoo, J. *Clustered objects*. PhD thesis, University of Toronto, 2005. AAINR07602.

[9] Appavoo, J., Auslander, M., Butrico, M., da Silva, D. M., Krieger, O., Mergen, M. F., Ostrowski, M., Rosenburg, B., Wisniewski, R. W., and Xenidis, J. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal 44* (January 2005), 427–440.

[10] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems 25* (August 2007).

[11] ARON, M., AND DRUSCHEL, P. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems 18*, 3 (2000), 197–228.

[12] AZIMI, R., SOARES, L., STUMM, M., WALSH, T., AND BROWN, A. D. PATH: Page Access Tracking to Improve Memory Management. In *Proceedings of the 6th international symposium on Memory management* (2007), ISMM '07, ACM, pp. 31–42.

[13] AZIMI, R., STUMM, M., AND WISNIEWSKI, R. W. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th Annual International Conference on Supercomputing* (2005), ICS '05, pp. 101–110.

[14] BAER, J.-L., AND CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1991), Supercomputing '91, ACM, pp. 176–186.

[15] BAILEY, D., HARRIS, T., SAPHIR, W., VAN DER WIJINGAART, R., WOO, A., AND YARROW, M. The NAS parallel benchmarks 2.0. Tech. Rep. NAS-95-020, NASA, 1995.

[16] BANGA, G., AND MOGUL, J. C. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (1998), USENIX ATC '98, pp. 1–12.

[17] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1999), USENIX Association, pp. 253–266.

[18] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[19] BARROSO, L. A., GHARACHORLOO, K., AND BUGNION, E. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (1998), ISCA '98, pp. 3–14.

[20] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles* (2009), SOSP '09, pp. 29–44.

[21] Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., and Isaacs, R. Your computer is already a distributed system: Why isn't your OS? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (2009), HotOS'09.

[22] Bellosa, F. Follow-on scheduling: Using TLB information to reduce cache misses. In *Sixteenth Symposium on Operating Systems Principles (SOSP '97), Work in Progress Session* (1997).

[23] Bellosa, F., and Steckermeier, M. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel Distrib. Comput. 37* (August 1996), 113–121.

[24] Berg, E., and Hagersten, E. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2004), ISPASS '04, IEEE Computer Society, pp. 20–27.

[25] Berg, E., and Hagersten, E. Fast data-locality profiling of native execution. In *Intl. Conf. on Measurement and Modelling of Computer Systems* (2005), SIGMETRICS'05, pp. 169–180.

[26] Bershad, B. N. The increasing irrelevance of IPC performance for micro-kernel-based operating systems. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures* (1992), pp. 205–212.

[27] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst. 9* (May 1991), 175–198.

[28] Bershad, B. N., Lee, D., Romer, T. H., and Chen, J. B. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), ASPLOS-VI, pp. 158–170.

[29] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), SOSP '95, ACM, pp. 267–283.

[30] Bhatia, S., Consel, C., and Lawall, J. Memory-manager/scheduler co-design: optimizing event-driven servers to improve cache behavior. In *Proceedings of the 5th International Symposium on Memory Management* (2006), ISMM '06, pp. 104–114.

[31] Bhattacharya, S., Pratt, S., Pulavarty, B., , and Morgan, J. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Ottawa Linux Symposium* (2003), pp. 371–386.

[32] Black, B., Annavaram, M., Brekelbaum, N., DeVale, J., Jiang, L., Loh, G. H., McCaule, D., Morrow, P., Nelson, D. W., Pantuso, D., Reed, P., Rupley, J., Shankar, S., Shen, J., and Webb,

C. Die stacking (3D) microarchitecture. In *Proceedings of the 39th Annual ACM/IEEE international symposium on Microarchitecture* (2006), MICRO-39, pp. 469 –479.

[33] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (2008), OSDI '08.

[34] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI '10, pp. 1–16.

[35] BRECHT, T., JANAKIRAMAN, G. J., LYNN, B., SALETORE, V., AND TURNER, Y. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), EuroSys '06, pp. 265–278.

[36] BROWN, Z. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium* (2007), OLS '07, pp. 81–85.

[37] BUGNION, E., ANDERSON, J. M., MOWRY, T. C., ROSENBLUM, M., AND LAM, M. S. Compiler-directed page coloring for multiprocessors. In *7th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS)* (1996), pp. 244–255.

[38] CAIN, H. W., AND NAGPURKAR, P. Runahead execution vs. conventional data prefetching in the IBM POWER6 microprocessor. In *Proceedings of ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software* (2010), pp. 203–212.

[39] CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1998), ASPLOS-VIII, ACM, pp. 139–149.

[40] CANTIN, J. F., AND HILL, M. D. Cache performance for selected SPEC CPU2000 benchmarks. *SIGARCH Comput. Archit. News 29* (September 2001), 13–18.

[41] CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. Compiler optimizations for improving data locality. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1994), ASPLOS-VI, ACM, pp. 252–262.

[42] CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 283–292.

[43] CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th Intl. Symp. on High-Performance Computer Architecture (HPCA)* (2005), pp. 340–351.

[44] CHAUDHURI, M. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 401–412.

[45] CHEN, J. B., AND BERSHAD, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)* (1993), pp. 120–133.

[46] 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. 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* (2006), ASID '06, pp. 63–65.

[47] CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, pp. 377–388.

[48] CHERITON, D. R. An experiment using registers for fast message-based interprocess communication. *SIGOPS Oper. Syst. Rev. 18* (October 1984), 12–20.

[49] CHI, C.-H., AND DIETZ, H. Improving cache performance by selective cache bypass. In *Twenty-Second Annual Hawaii International Conference on System Sciences* (1989), vol. 1, Architecture Track, pp. 277–285.

[50] CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 13–24.

[51] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Making pointer-based data structures cache conscious. *Computer 33*, 12 (2000), 67–74.

[52] CHO, S., AND JIN, L. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Intl. Symp. on Microarchitecture (MICRO)* (2006), pp. 455–468.

[53] CHUKHMAN, I., AND PETROV, P. Context-aware TLB preloading for interference reduction in embedded multi-tasked systems. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI* (2010), GLSVLSI '10, ACM, pp. 401–404.

[54] CLARK, D. W. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems 1* (February 1983), 24–37.

[55] COLLINS, J. D., AND TULLSEN, D. M. Hardware identification of cache conflict misses. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture* (1999), IEEE Computer Society, pp. 126–135.

[56] COLMENARES, J. A., BIRD, S., COOK, H., PEARCE, P., ZHU, D., SHALF, J., HOFMEYR, S., ASANOVIÄĞ, K., AND KUBIATOWICZ, J. Resource management in the Tessellation manycore OS. In *Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism* (2010), HotPar'10.

[57] COMMITTEE, I. R. International technology roadmap for semiconductors, 2010 update. http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/2010_Update_Overview.pdf, accessed on July 2011.

[58] CORBATÃŞ, F. J. A paging experiment with the Multics system. *In Honor of Philip M. Morse* (1968), 217–225.

[59] DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *30th Intl. Symp. on Microarchitecture (MICRO)* (1997), pp. 292–302.

[60] DENNING, P. J. The working set model for program behavior. *Communications of the ACM 11* (May 1968), 323–333.

[61] DENNING, P. J. Virtual memory. *ACM Computing Surveys (CSUR) 2* (September 1970), 153–189.

[62] DENNING, P. J. The locality principle. *Commun. ACM 48* (July 2005), 19–24.

[63] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), PLDI '03, ACM, pp. 245–257.

[64] DRAVES, R. P., BERSHAD, B. N., RASHID, R. F., AND DEAN, R. W. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (1991), SOSP '91, pp. 122–136.

[65] DREPPER, U., AND MOLNAR, I. The Native POSIX Thread Library for Linux. Tech. rep., RedHat Inc, 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[66] DRONGOWSKI, P. J. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. http://developer.amd.com/assets/amd_ibs_paper_en.pdf (retrieved Aug/2011), 2007.

[67] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1996), pp. 261–275.

[68] DYBDAHL, H., AND STENSTRÖM, P. Enhancing last-level cache performance by block bypassing and early miss determination. In *Asia-Pacific Computer Systems Arch. Conf.* (2006), pp. 52–66.

[69] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2004), pp. 21–21.

[70] ENGLER, D. R., AND KAASHOEK, M. F. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 78–.

[71] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[72] ERLINGSSON, U., VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 6–6.

[73] ETSION, Y., AND FEITELSON, D. G. L1 cache filtering through random selection of memory references. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (2007), PACT '07, IEEE Computer Society, pp. 235–244.

[74] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT* (2007), IEEE Computer Society, pp. 25–38.

[75] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal* (2004).

[76] FOONG, A. P., HUFF, T. R., HUM, H. H., PATWARDHAN, J. R., AND REGNIER, G. J. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 70–79.

[77] FOTHERINGHAM, J. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Commun. ACM 4* (October 1961), 435–436.

[78] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1999), FOCS '99, IEEE Computer Society, pp. 285–.

[79] FU, J. W. C., AND PATEL, J. H. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th annual international symposium on Computer architecture* (1991), ISCA '91, ACM, pp. 54–63.

[80] FULLER, S. H., AND MILLETT, L. I. Computing performance: Game over or next level? *Computer 44* (2011), 31–38.

[81] FULLER, S. H., AND MILLETT, L. I. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[82] GAMMO, L., BRECHT, T., SHUKLA, A., AND PARIAG, D. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of 6th Annual Ottawa Linux Symposium* (2004), pp. 215–225.

[83] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 87–100.

[84] GAMSA, B., KRIEGER, O., AND STUMM, M. Optimizing IPC performance for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01* (Washington, DC, USA, 1994), ICPP '94, IEEE Computer Society, pp. 208–211.

[85] GONZÁLEZ, A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In *Intl. Conf. in Supercomputing (ICS)* (1995), pp. 338–347.

[86] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture - A quantitative approach, 3rd Edition*. Morgan Kaufmann, 2003.

[87] HILL, M. D., AND SMITH, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38* (December 1989), 1612–1630.

[88] HOROWITZ, M., ALON, E., PATIL, D., NAFFZIGER, S., KUMAR, R., AND BERNSTEIN, K. Scaling, power and the future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems* (2007), VLSID '07, IEEE Computer Society, pp. 23–.

[89] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review 41* (April 2007), 37–49.

[90] INTEL. *Intel 64 and IA-32 Architectures Software DeveloperâĂŹs Manual, Volume 3 (3A and 3B): System Programming Guide*. Intel Corporation, May 2011.

[91] IWAI, H. Roadmap for 22-nm and beyond. *Microelectronic Engineering 86*, 7-9 (2009), 1520 – 1528. INFOS 2009.

[92] Jacob, B. L., and Mudge, T. N. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1998), ASPLOS-VIII, ACM, pp. 295–306.

[93] Jaleel, A. Memory characterization of workloads using instrumentation-driven simulation. http://www.glue.umd.edu/ ajaleel/workload/, Retrieved Sep, 2011.

[94] Jaleel, A., Cohn, R. S., keung Luk, C., and Jacob, B. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on CMPs. Tech. rep., 2006.

[95] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, Jr., S., and Emer, J. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 208–219.

[96] Jaleel, A., Theobald, K. B., Steely, Jr., S. C., and Emer, J. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th annual international symposium on Computer architecture* (2010), ISCA '10, ACM, pp. 60–71.

[97] Jiang, S., and Zhang, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[98] Johnson, T., and Shasha, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 439–450.

[99] Johnson, T. L., Connors, D. A., Merten, M. C., and mei W. Hwu, W. Run-time cache bypassing. *IEEE Transactions on Computers 48*, 12 (1999), 1338–1354.

[100] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture* (1990), ISCA '90, ACM, pp. 364–373.

[101] Kandemir, M., Yemliha, T., Muralidhara, S., Srikantaiah, S., Irwin, M. J., and Zhnag, Y. Cache topology aware computation mapping for multicores. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 74–85.

[102] KANDIRAJU, G. B., AND SIVASUBRAMANIAM, A. Going the distance for TLB prefetching: An application-driven study. In *Proceedings of the 29th annual international symposium on Computer architecture* (2002), ISCA '02, IEEE Computer Society, pp. 195–206.

[103] KENNEDY, K., AND MCKINLEY, K. S. Optimizing for parallelism and data locality. In *Proceedings of the 6th international conference on Supercomputing* (New York, NY, USA, 1992), ICS '92, ACM, pp. 323–334.

[104] KESSLER, R. E., AND HILL, M. D. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems 10*, 4 (1992), 338–359.

[105] KHAN, S. M., JIMÉNEZ, D. A., BURGER, D., AND FALSAFI, B. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), PACT '10, ACM, pp. 489–500.

[106] KHARBUTLI, M., AND SOLIHIN, Y. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers 57*, 4 (2008), 433–447.

[107] KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., AND SUMNER, F. H. One-level storage system. *Electronic Computers, IRE Transactions on EC-11*, 2 (april 1962), 223 –235.

[108] KNICKERBOCKER, J., ANDRY, P., DANG, B., HORTON, R., PATEL, C., POLASTRE, R., SAKUMA, K., SPROGIS, E., TSANG, C., WEBB, B., AND WRIGHT, S. 3D silicon integration. In *Electronic Components and Technology Conference, 2008. ECTC 2008. 58th* (2008), pp. 538 –543.

[109] KNOWLTON, K. C. A fast storage allocator. *Communications of the ACM 8* (October 1965), 623–624.

[110] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (2007), USENIX Association, pp. 7:1–7:14.

[111] LAM, M. D., ROTHBERG, E. E., AND WOLF, M. E. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1991), ASPLOS-IV, ACM, pp. 63–74.

[112] LARUS, J., AND PARKES, M. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2002), pp. 103–114.

[113] LEMON, J. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), USENIX Association, pp. 141–153.

[114] Li, T., John, L. K., Sivasubramaniam, A., Vijaykrishnan, N., and Rubio, J. Understanding and Improving Operating System Effects in Control Flow Prediction. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2002), pp. 68–80.

[115] Liedtke, J. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 175–188.

[116] Liedtke, J. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[117] Liedtke, J. Colorable memory. Nov. 1996.

[118] Liedtke, J., Hartig, H., and Hohmuth, M. OS-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium* (1997), pp. 213–227.

[119] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In *14th Intl. Symp. on High-Performance Comp. Arch. (HPCA)* (2008), pp. 367–378.

[120] Lin, W., and Reinhardt, S. Predicting last-touch references under optimal replacement. Tech. Rep. CSE-TR-447-02, University of Michigan, 2002.

[121] Lin, W.-F., Reinhardt, S. K., Burger, D., and Puzak, T. R. Filtering superfluous prefetches using density vectors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 124–132.

[122] Liptay, J. S. Structural aspects of the System/360 Model 85 II: The cache. *IBM Systems Journal 7*, 1 (1968), 15–21.

[123] Liu, H., Ferdman, M., Huh, J., and Burger, D. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture* (2008), MICRO 41, IEEE Computer Society, pp. 222–233.

[124] Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., and Kubiatowicz, J. Tessellation: space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX Workshop on Hot topics in parallelism* (Berkeley, CA, USA, 2009), HotPar'09, USENIX Association, pp. 10–10.

[125] Lynch, W. L., Bray, B. K., and Flynn, M. J. The effect of page allocation on caches. In *25th Intl. Symp. on Microarchitecture (MICRO)* (1992), pp. 222–225.

[126] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35*, 2 (2002), 50–58.

[127] Marko, M., and Madison, A. W. Cache conflict resolution through detection, analysis and dynamic remapping of active pages. In *ACM-SE 38: Proceedings of the 38th annual on Southeast regional conference* (2000), ACM, pp. 60–66.

[128] Maynard, A. M. G., Donnelly, C. M., and Olszewski, B. R. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1994), ASPLOS-VI, ACM, pp. 145–156.

[129] McCurdy, C., Cox, A. L., and Vetter, J. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software* (2008), IEEE Computer Society, pp. 95–104.

[130] Mogul, J. C., and Borg, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991), pp. 75–84.

[131] Mogul, J. C., Mudigonda, J., Binkert, N., Ranganathan, P., and Talwar, V. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28*, 3 (2008), 26–41.

[132] Mogul, J. C., and Ramakrishnan, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15* (August 1997), 217–252.

[133] Moore, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr. 1965), 114–117.

[134] Moore, G. E. Progress in digital integrated electronics. In *International Electron Devices Meeting* (1975), vol. 21, pp. 11–13.

[135] Mosberger, D., and Jin, T. httperf – A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev. 26* (December 1998), 31–37.

[136] Mueller, F. Compiler support for software-based cache partitioning. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (1995), pp. 125–133.

[137] Murray, T. J., Madison, A. W., and Westall, J. M. Lookahead page placement. In *ACM-SE 33: Proceedings of the 33rd annual on Southeast regional conference* (1995), ACM, pp. 146–155.

[138] Mutlu, O., Kim, H., Armstrong, D. N., and Patt, Y. N. Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 2–9.

[139] MYSORE, S., MAZLOOM, B., AGRAWAL, B., AND SHERWOOD, T. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (2008), ASPLOS XIII, pp. 211–221.

[140] NELLANS, D., BALASUBRAMONIAN, R., AND BRUNVAND, E. OS execution on multi-cores: is outsourcing worthwhile? *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 104–105.

[141] NELLANS, D., SUDAN, K., BRUNVAND, E., AND BALASUBRAMONIAN, R. Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionality. In *Sixth Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2010), pp. 13–20.

[142] NEUMANN, J. v. First draft of a report on the EDVAC. Tech. rep., 1945.

[143] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., AND HUNT, G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, pp. 221–234.

[144] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: an efficient and portable web server. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1999), USENIX Association, pp. 15–15.

[145] PALACHARLA, S., AND KESSLER, R. E. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture* (Los Alamitos, CA, USA, 1994), ISCA '94, IEEE Computer Society Press, pp. 24–33.

[146] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of Web server architectures. In *Proceedings of the 2nd European Conference on Computer Systems (Eurosys)* (2007), pp. 231–243.

[147] PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 335–348.

[148] PETOUMENOS, P., KERAMIDAS, G., ZEFFER, H., KAXIRAS, S., AND HAGERSTEN, E. Modeling cache sharing on chip multiprocessor architectures. In *Intl. Symp. on Workload Characterization (IISWC)* (2006), pp. 160–171.

[149] PIQUET, T., ROCHECOUSTE, O., AND SEZNEC, A. Exploiting single-usage for effective memory management. In *Asia-Pacific Computer Systems Architecture Conference* (2007), pp. 90–101.

[150] PRABHAT JAIN, SRINI DEVADAS, L. R. Controlling cache pollution in prefetching with software-assisted cache replacement. Tech. Rep. CSG-462, MIT, 2001.

[151] Provos, N. libevent - An Event Notification Library. http://www.monkey.org/~provos/libevent.

[152] Przybylski, S. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th annual international symposium on Computer Architecture* (1990), ISCA '90, ACM, pp. 160–169.

[153] Purohit, A., Wright, C. P., Spadavecchia, J., and Zadok, E. Cosy: develop in user-land, run in kernel-mode. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), USENIX Association, pp. 19–19.

[154] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. In *Intl. Symp. on Comp. Arch. (ISCA)* (2007), pp. 381–391.

[155] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 423–432.

[156] Rafique, N., Lim, W.-T., and Thottethodi, M. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 2–12.

[157] Rajagopalan, M., Debray, S. K., Hiltunen, M. A., and Schlichting, R. D. Cassyopia: compiler assisted system optimization. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003), pp. 18–18.

[158] Rangarajan, M., Bohra, A., Banerjee, K., Carrera, E. V., Bianchini, R., Iftode, L., and Zwaenepoel, W. TCP Servers: Offloading TCP processing in internet servers. design, implementation, and performance. Tech. rep., Rutgers University, 2002.

[159] Reddy, R., and Petrov, P. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (2007), pp. 198–207.

[160] Redstone, J. A., Eggers, S. J., and Levy, H. M. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000), pp. 245–256.

[161] Reese, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* (2008).

[162] Reynolds, J. C. The discoveries of continuations. *Lisp Symb. Comput. 6* (November 1993), 233–248.

[163] Rivers, J. A., and Davidson, E. S. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP, Vol. 1* (1996), pp. 154–163.

[164] Romer, T. H., Lee, D., and Bershad, B. N. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 1994), USENIX Assoc., pp. 255–266.

[165] Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J. Enabling scalability and performance in a large scale CMP environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, pp. 73–86.

[166] Sanchez, D., and Kozyrakis, C. The ZCache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), MICRO '43, IEEE Computer Society, pp. 187–198.

[167] Saulsbury, A., Dahlgren, F., and Stenström, P. Recency-based TLB Preloading. In *Proceedings of the 27th annual international symposium on Computer architecture* (2000), ISCA '00, ACM, pp. 117–127.

[168] Sayre, D. Is automatic "folding" of programs efficient enough to displace manual? *Commun. ACM 12* (December 1969), 656–660.

[169] Schneider, F. T., Payer, M., and Gross, T. R. Online optimizations driven by hardware performance monitoring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI'07, pp. 373–382.

[170] Schroeder, B., Wierman, A., and Harchol-Balter, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3* (2006), NSDI'06, USENIX Association, pp. 18–18.

[171] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), OSDI '96, ACM, pp. 213–227.

[172] Sen, S., Chatterjee, S., and Dumir, N. Towards a theory of cache-efficient algorithms. *J. ACM 49* (November 2002), 828–858.

[173] Shalev, L., Satran, J., Borovik, E., and Ben-Yehuda, M. IsoStack: highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC'10, USENIX Association, pp. 5–5.

[174] Shen, X., Shaw, J., Meeker, B., and Ding, C. Locality approximation using time. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 55–61.

[175] Sherwood, T., Calder, B., and Emer, J. Reducing cache misses using hardware and software page placement. In *International Conference on Supercomputing (ICS)* (1999), pp. 155–164.

[176] Sherwood, T., Sair, S., and Calder, B. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (New York, NY, USA, 2000), MICRO 33, ACM, pp. 42–53.

[177] Sites, R. L., and Agarwal, A. Multiprocessor cache analysis using ATUM. In *International Symposium on Computer Architecture (ISCA)* (1988), pp. 186–195.

[178] Small, C., and Seltzer, M. A comparison of OS extension technologies. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), USENIX Association, pp. 4–4.

[179] Smith, A. J. Sequential program prefetching in memory hierarchies. *Computer 11* (December 1978), 7–21.

[180] Smith, A. J. Cache memories. *ACM Computing Surveys (CSUR) 14* (September 1982), 473–530.

[181] Smith, A. J. Design of CPU cache memories. Tech. Rep. UCB/CSD-87-357, EECS Department, University of California, Berkeley, Jun 1987.

[182] Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A. Spatial memory streaming. In *Proceedings of the 33rd annual international symposium on Computer Architecture* (Washington, DC, USA, 2006), ISCA '06, IEEE Computer Society, pp. 252–263.

[183] Sprunt, B. Pentium 4 performance-monitoring features. *IEEE Micro 22* (July 2002), 72–82.

[184] Srinivasan, V., Davidson, E. S., and Tyson, G. S. A prefetch taxonomy. *IEEE Trans. Comput. 53* (February 2004), 126–140.

[185] Suh, G. E., Rudolph, L., and Devadas, S. Dynamic partitioning of shared cache memory. *Journal of Supercomputing 28*, 1 (2004), 7–26.

[186] Tam, D., Azimi, R., Soares, L., and Stumm, M. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2007).

[187] Tam, D., Azimi, R., and Stumm, M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), ACM, pp. 47–58.

[188] Tam, D. K., Azimi, R., Soares, L. B., and Stumm, M. Rapidmrc: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS '09, ACM, pp. 121–132.

[189] Tiwari, M., Wassel, H. M., Mazloom, B., Mysore, S., Chong, F. T., and Sherwood, T. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (2009), ASPLOS '09, pp. 109–120.

[190] Tse, J., and Smith, A. J. CPU cache prefetching: Timing evaluation of hardware implementations. *IEEE Trans. Comput. 47* (May 1998), 509–526.

[191] Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A. R. A modified approach to data cache management. In *28th Intl. Symp. on Microarchitecture (MICRO)* (1995), pp. 93–103.

[192] Vera, X., Lisper, B., and Xue, J. Data caches in multitasking hard real-time systems. In *24th IEEE International Real-Time Systems Symposium (RTSS)* (2003), pp. 154–165.

[193] VMWare. *VMWare Virtual Machine Interface Specification*. http://www.vmware.com/pdf/vmi_specs.pdf.

[194] von Behren, R., Condit, J., and Brewer, E. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003).

[195] von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 268–281.

[196] Wang, D. T. *Modern DRAM memory systems: performance analysis and scheduling algorithm*. PhD thesis, 2005. AAI3178628.

[197] Weissman, B. Performance counters and state sharing annotations: a unified approach to thread locality. In *ASPLOS-VIII* (1998), ACM, pp. 127–138.

[198] Welsh, M., Culler, D., and Brewer, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001), SOSP '01, pp. 230–243.

[199] Wentzlaff, D., and Agarwal, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 76–85.

[200] Wilkes, M. V. The memory gap and the future of high performance memories. *SIGARCH Computer Architecture News 29* (March 2001), 2–7.

[201] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (New York, NY, USA, 1991), PLDI '91, ACM, pp. 30–44.

[202] WOLFE, A. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering 2*, 3 (1994), 315–327.

[203] WONG, W. A., AND BAER, J.-L. Modified LRU policies for improving second-level cache behavior. In *6th Intl. Symp. on High-Performance Comp. Arch. (HPCA)* (2000), pp. 49–60.

[204] WU, Y., RAKVIC, R., CHEN, L.-L., MIAO, C.-C., CHRYSOS, G., AND FANG, J. Compiler managed micro-cache bypassing for high performance EPIC processors. In *Intl. Symp. on Microarchitecture (MICRO)* (2002), pp. 134–145.

[205] XIANG, L., CHEN, T., SHI, Q., AND HU, W. Less reused filter: improving L2 cache performance via filtering less reused lines. In *Proceedings of the 23rd international conference on Supercomputing* (2009), ICS '09, ACM, pp. 68–79.

[206] ZEBCHUK, J., SAFI, E., AND MOSHOVOS, A. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *40th Intl. Symp. on Microarchitecture (MICRO)* (2007), pp. 314–327.

[207] ZELDOVICH, N., YIP, A., DABEK, F., MORRIS, R., MAZIÈRES, D., AND KAASHOEK, F. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX)* (June 2003).

[208] ZHANG, Y., KANDEMIR, M., AND YEMLIHA, T. Studying inter-core data reuse in multicores. *SIGMETRICS Perform. Eval. Rev. 39* (June 2011), 25–36.

[209] ZHOU, Y., PHILBIN, J., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 91–104.

[210] ZHUANG, X., AND LEE, H.-H. S. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Trans. Comput. 56* (January 2007), 18–31.