Operating System Management of Shared Caches on Multicore Processors

by

David Tam

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

Abstract

Operating System Management of Shared Caches on Multicore Processors

David Tam

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2010

Our thesis is that operating systems should manage the on-chip shared caches of multicore processors for the purposes of achieving performance gains. Consequently, this dissertation demonstrates how the operating system can profitably manage these shared caches. Two shared-cache management principles are investigated: (1) promoting shared use of the shared cache, demonstrated by an automated online thread clustering technique, and (2) providing cache space isolation, demonstrated by a software-based cache partitioning technique. In support of providing isolation, cache provisioning is also investigated, demonstrated by an automated online technique called RapidMRC. We show how these software-based techniques are feasible on existing multicore systems with the help of their hardware performance monitoring units and their associated hardware performance counters. On a 2-chip IBM POWER5 multicore system, promoting sharing reduced processor pipeline stalls caused by cross-chip cache accesses by up to 70%, resulting in performance improvements of up to 7%. On a larger 8-chip IBM POWER5+ multicore system, the potential for up to 14% performance improvement was measured. Providing isolation improved performance by up to 50%, using an exhaustive offline search method to determine optimal partition size. On the other hand, up to 27% performance improvement was extracted from the corresponding workload using an automated online approximation technique, made possible by RapidMRC.

Acknowledgements

"If I have seen further, it is by standing on the shoulders of giants." – Isaac Newton

I am grateful to have been acquainted with some of my forefathers in the field of computer systems research, who have done pioneering work in operating system performance scalability: Orran Krieger, Ben Gamsa, Jonathan Appavoo, the University of Toronto Hurricane/Tornado/K42 OS group, and the IBM K42 OS group. It is their work that has led me to my research path.

This dissertation has been made possible by the generous support of many individuals. There are three individuals, in particular, whom I will forever be indebted. My supervisor, Professor Michael Stumm, has provided incomparable academic guidance and wisdom. My colleague, Reza Azimi, has made this dissertation possible; I *cannot* imagine how I could have done it without him. My colleague, Livio Soares, has been a vital source of inspiration with his positive outlook, boundless energy, humour, and *chutzpah*. I feel that the four of us together, as a paper publishing team, have conquered the world.

I would like to thank the other members of my research group, who have all enriched my Ph.D. graduate school experience, accompanying my years with intelligent and unique minds and personalities: Adam Czajkowski, Alex Depoutovitch, Raymond Fingas, Moshe Krieger, Maria Leitner, Adrian Tam, and Tom Walsh.

I am fortunate to have been aided by numerous individuals, external to my research group, during the production of this dissertation. I would like to thank: Allan Kielstra, Orran Krieger, Dilma Da Silva, Jimi Xenidis, and Santosh Rao, all five from IBM; Professor Cristiana Amza and Gokul Soundararajan, both at the University of Toronto; Professors Greg Steffan, Ashvin Goel, Angela Demke Brown, and Michael Voss, who are my departmental thesis committee members; and Professor Gernot Heiser, who is my external thesis committee member.

In terms of life support, I would like to thank my wife, Jennie Fan, for her patience and understanding. What seemed like eternity has finally passed. I am grateful to my parents for providing a supportive environment that has enabled me to reach this point in life's journey.

Finally, I would like to acknowledge my sources of financial support: the Edward S. Rogers Sr Department of Electrical and Computer Engineering at the University of Toronto, the Edward S. Rogers Sr National Electrical and Computer Engineering Scholarship Program, and the Government of Ontario through the Ontario Graduate Scholarship Program. This work was also supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

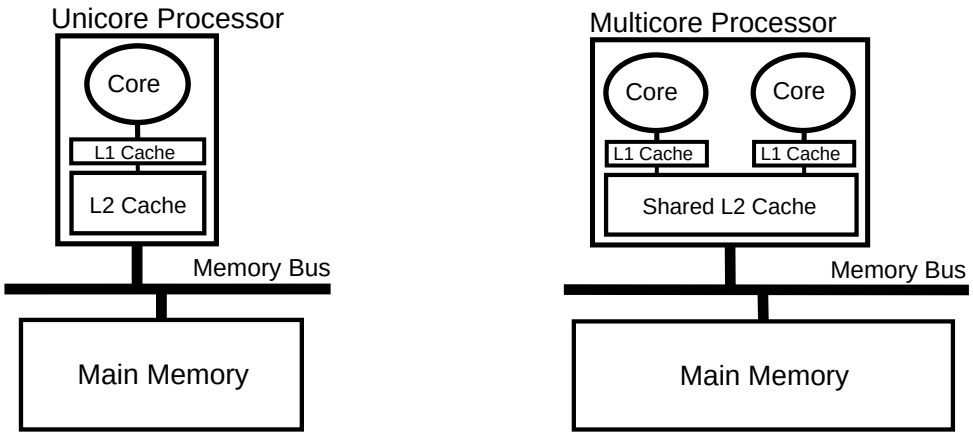
"Genius is one percent inspiration and ninety-nine percent perspiration." – Thomas Edison

A primary responsibility of operating systems is to manage the limited, shared, hardware resources of a computer and do so in a low-overhead manner. Traditionally, these resources included microprocessors (processors), main memory, disks, and network cards. Managing these resources intelligently in order to maximize performance or ensure fairness is a classic theme in operating systems research.

Due to the large speed disparity between processors and main memory, caches have been incorporated into processors. These on-chip caches are small but high-speed memories that serve as temporary buffers to larger but slower off-chip main memory. The on-chip caches are a performance-critical hardware resource because they can reduce data access latencies by an order of magnitude when compared to off-chip main memory access. For example, on the IBM POWER5 processor, accessing data that is located in the on-chip L2 cache requires 14 processor cycles, whereas accessing data that is located in main memory requires 280 cycles.

The recent arrival of multiple-core (multicore) processors has changed the hardware landscape considerably in that they have introduced on-chip caches that are *shared* by multiple execution cores. In contrast, previous generation single-core (unicore) processors featured an on-chip *private* cache that directly served only a single execution core. This introduction of sharing into such a performance-critical hardware resource can have a significant impact on performance. Figure 1.1a illustrates a simplified view of a traditional unicore processor, while Figure 1.1b illustrates a simplified view of a multicore processor.

In processors of today, there are typically 2 or 3 levels of on-chip cache. The level 1 (L1) cache is closest to the processor execution core and has the shortest access time but the lowest capacity. The level 2 (L2) cache is next in the cache hierarchy and has longer access times but larger capacity. Finally, there may be a level 3 (L3) cache with even longer access times but even larger capacity. On multicore processors, the last-level cache, which is the level before requiring off-chip main memory access, is typically shared among several cores. Typically, this component is the L2 cache or the



a. unicore processor system b. multicore processor system

Figure 1.1: Simplified views of a traditional single-core (unicore) processor and a multiple-core (multicore) processor system.

L3 cache.

Although 2-core and 4-core processors are widely available today, the number of cores in future processors will only increase due to Moore's law, which predicts that the number of transistors on an integrated circuit will double approximately every 2 years [Moore 1965 1975]. For example, IBM recently began shipping the 8-core POWER7 processor; Sun currently offers the 8-core UltraSPARC T2 processor with future plans for a 16-core UltraSPARC T3 processor; AMD has a 6-core Opteron processor with plans for a 16-core processor; and Intel recently released an 8-core Xeon processor. Even game consoles use multicore processors. The Sony PLAYSTATION 3 uses the 9-core Sony-Toshiba-IBM Cell processor, while the Microsoft Xbox 360 features a 3-core Apple-IBM-Motorola PowerPC Xenon processor.

With the eminent ubiquity of multicore processors, we contend that their performance-critical on-chip shared caches are a hardware resource that should be intelligently managed by the operating system. On-chip caches have a limited size and therefore are a scarce, finite resource. Despite continual increases in the size of on-chip caches, the demand for cache space has always exceeded its supply because as on-chip cache sizes and the number of execution cores increase, so do application problem sizes. Future multicore processors will feature higher degrees of on-chip cache sharing, which may increase cache contention considerably and thus critically require operating system management. In this dissertation, we demonstrate how managing these shared caches at the operating system level can result in application performance improvements when compared to traditional operating systems that are not *"shared cache"*-aware. In managing shared caches at the operating system level, we address the challenges of maintaining low overhead and developing software-only techniques that are deployable on existing multicore processors of today.

We explore two management principles[1] to control how a shared cache is utilized by applications,

[1] A definition of *principle*, according to the Oxford English Dictionary is, *"**II.** Fundamental truth or law; motive force. **3. a.** A fundamental truth or proposition on which others depend; a general statement or tenet forming the (or a) basis of a system of belief, etc.; a primary assumption forming the basis of a chain of reasoning."*

which correspond to maximizing a major advantage and minimizing a major disadvantage of on-chip shared caches. The first management principle *promotes sharing* of the cache by identifying and facilitating shared usage among threads or processes. Once sharing is identified, threads or processes can be scheduled in a manner that maximizes the occurrence of sharing within the on-chip shared cached. For example, threads or processes that share data can be co-scheduled to all run at the same time so that they can exploit the shared cache for sharing. As another example, on a computer system consisting of *multiple* processor chips, where each chip is a multicore processor, threads or processes that share data across processor chips can be migrated onto the same chip to induce shared use of the local on-chip shared cache. Multiple threads of execution, which share a single application address space, can intimately share a lot of data. In addition, multiple processes with separate address spaces can often share common code libraries. They may also have regions of their address space explicitly configured for shared access across processes. The first management principle exploits a major advantage of a shared cache, namely its fast on-chip sharing capabilities. Logical sharing at the software level, among threads or processes, can be mapped to a physically shared resource at the hardware level, resulting in fast sharing.

The second management principle *provides isolation* in the shared cache by preventing non-sharing threads or processes from utilizing the same sections of the cache and interfering with each other's cache occupancy. In effect, a large shared cache can be partitioned into several smaller private caches. This principle circumvents a major disadvantage of shared caches, namely cache space interference among applications, where non-sharing threads or processes can inadvertently evict each other's data from the cache, leading to performance degradation.

In traditional unicore processors and operating systems, these two management principles were not applicable to the on-chip caches since they were private rather than shared among multiple execution cores. However, these principles were applied in the past to main memory, since it can be viewed as a shared resource. For example, the first principle of promoting sharing can be seen in NUMA (non-uniform memory access latency) multiprocessor systems research. Traditionally, NUMA systems consist of multiple unicore processors that are grouped together in a hierarchical manner, with each group (*node*) containing local memory and the ability to access remote memory belonging to remote nodes. NUMA systems researchers have explored how to actively encourage sharing within the fast local memory of a node rather than allow sharing across slow remote memory of remote nodes [Appavoo et al. 2007; Bellosa and Steckermeier 1996; Gamsa et al. 1999]. Analogously in the realm of multicore systems, sharing at the software level among threads or processes can be mapped to an efficient shared resource at the hardware level, resulting in fast sharing capabilities.

The second principle of providing isolation can be seen in memory page or buffer management research that explores local policies as an alternative to global policies in an effort to reduce physical memory space interference between competing applications [Azimi et al. 2007; Choi et al. 2000; Soundararajan et al. 2008]. For example, in managing physical pages of a virtual memory system,

researchers have explored local per-region page replacement policies as an alternative to global page replacement policies [Azimi et al. 2007]. Analogously in the realm of multicore processors, preventing hardware resource interference among non-sharing, disparate threads or processes can result in performance improvements.

When providing cache space isolation to an application, cache space must be allocated to the application. Determining how much cache space to allocate to each application is another management role of the operating system. This task is typically known as *resource provisioning* and is required for scarce hardware resources, such as the on-chip shared cache. In this dissertation, we also explore how the operating system can determine the amount of on-chip cache space to allocate to each application.

1.1 Thesis and Goals of Dissertation

Our thesis is that operating systems should manage the on-chip shared caches of multicore processors for the purposes of achieving performance gains. Consequently, the goal of this dissertation is to develop and demonstrate methodologies for how the operating system can profitably manage these shared caches. We develop and demonstrate methodologies to realize the two shared-cache management principles of *promoting sharing* and *providing isolation*. In support of providing isolation, we develop and demonstrate a methodology to provision the shared cache in an automated online manner.

1.2 Dissertation Outline

To establish the context of our research, Chapter 2 provides general background on the evolution of computer hardware and operating systems. The principle of promoting sharing is investigated in Chapter 3, followed by the principle of providing isolation in Chapter 4. Provisioning the shared cache in support of providing isolation is investigated in Chapter 5. General issues that relate to all three components are discussed in Chapter 6. Future work is described in Chapter 7. Finally, we conclude in Chapter 8.

We now provide a brief overview of the two shared-cache management principles and shared cache provisioning.

1.2.1 Promoting Sharing

The shared nature of on-chip caches is a property that can be exploited for performance gains. Data and instructions that are commonly accessed by all cores in a shared manner can be quickly reached by all cores. This hardware performance characteristic leads to our first principle of promoting sharing in the shared cache. An operating system scheduler could select processes or threads that

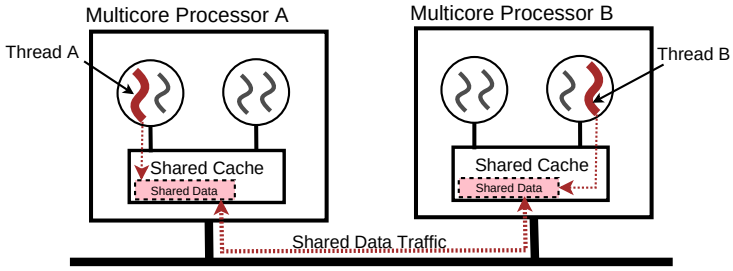


Figure 1.2: Promoting sharing within a shared cache. Thread `A` and thread `B` share data. Thread `B` could be migrated to multicore processor `A` so that the shared data is located within a single shared cache, resulting in faster access by both threads, leading to increased performance.

share data or instructions and co-schedule them to all run at the same time within the same multicore processor so that they can exploit the shared cache for sharing.

On a computer system consisting of multiple processor chips, each of which is a multicore processor, this goal involves migrating the sharing threads or processes, which may be scattered across several multicore chips, onto a single multicore chip. Figure 1.2 depicts a scenario where thread `B` could be migrated to multicore processor `A` so that the data shared between thread `A` and thread `B` can be located within a single shared cache. On a simpler computer system consisting of only a single multicore processor chip, thread or process migration is not applicable but simply co-scheduling the sharing threads or processes is sufficient to promote sharing. This scheme can also be applied to multiple processes, which don't share an address space but may share common code libraries. By promoting sharing, we exploit a major advantage of shared caches.

We develop and demonstrate a methodology that applies the principle of promoting sharing in Chapter 3. We target large multithreaded commercial workloads that execute in a single-programmed (single application) computing environment. On a small-scale multiple-chip platform[2], we reduce processor pipeline stalls caused by cross-chip cache accesses by up to 70%, resulting in performance improvements of up to 7%. Preliminary results on a larger-scale multiple-chip platform[3] indicate the potential for up to 14% performance improvement.

In summary, we use operating system scheduling to promote sharing within the shared cache. By promoting sharing, we attempt to maximize a major benefit of shared caches, namely its fast sharing capabilities. We match the sharing that occurs in software with the available hardware sharing facilities.

1.2.2 Providing Isolation

In some workload environments, sharing among processes or threads is not prevalent. An example would be a multiprogrammed, single-threaded computing environment, which consists of multiple applications that each have one thread of execution. In this situation, disparate processes, which

[2] An *8-way* IBM POWER5 system consisting of 2 chips × 2 cores per chip × 2 hardware threads per core.
[3] A *32-way* IBM POWER5+ system consisting of 8 chips × 2 cores per chip × 2 hardware threads per core.

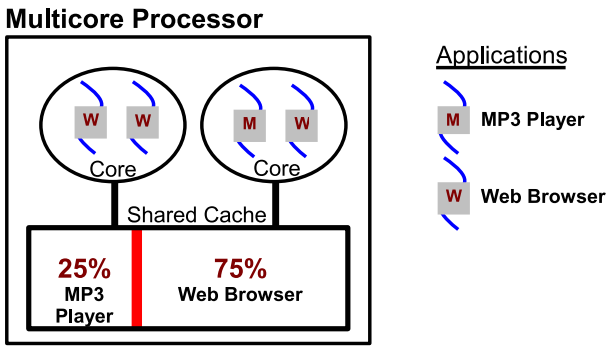


Figure 1.3: Providing isolation in a shared cache by partitioning it into smaller private caches. The MP3 player is allocated 25% of the L2 cache space for exclusive use, while the web browser is given the remaining 75%. This partitioning eliminates potential cache space interference between the two applications that would lead to performance degradation.

do not share any data or instructions, share the cache in an unrestricted manner and may interfere with each other when executing on the cores. Specifically, this situation can lead to cache line interference between non-sharing processes, resulting in significant performance degradation. A process may unintentionally evict cache lines belonging to a non-related process that is currently executing on another core of the processor rather than evict one of its own cache lines. Consider, for example, an MP3 player that streams through a lot of data without reuse. It severely and continuously pollutes the shared cache with an attendant negative effect on the performance of the other applications running on the other cores of the processor. This scenario of non-shared use of a shared hardware resource leads us to our second management principle of providing isolation in the shared cache. By providing cache space isolation, we circumvent a major disadvantage of shared caches.

We develop and demonstrate a methodology that applies the principle of providing isolation in Chapter 4. We control which processes or threads have access to which sections of the shared cache, as shown in Figure 1.3, and we can then effectively and flexibly partition the shared cache into smaller private caches. We accomplish this task at the operating system level by controlling the allocation of physical pages. We demonstrate performance improvements of up to 17% in terms of instructions-per-cycle (IPC).

In summary, we use operating system memory management to control the physical occupation of the shared cache in order to provide cache space isolation. By providing isolation, we attempt to minimize a major disadvantage of shared caches, namely cache space interference among applications.

1.2.3 Provisioning the Shared Cache

A fundamental requirement of providing isolation is knowing how much cache space to allocate to each application. For example, when partitioning the shared cache in Figure 1.3, the operating system must decide upon an appropriate size for each partition. One possible solution is to use the

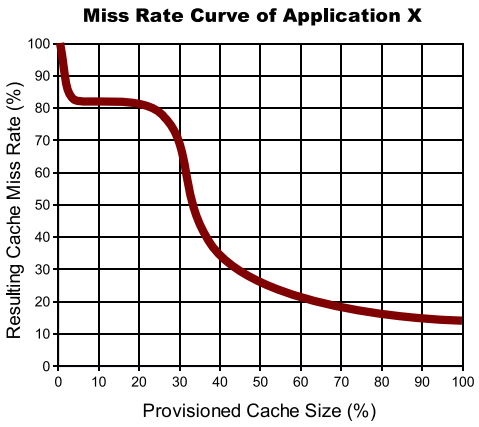


Figure 1.4: A cache miss rate curve showing the trade-off spectrum between provisioned cache size and the resulting cache miss rate. It characterizes the cache space requirements of the application and therefore can be used for cache provisioning.

trial-and-error technique of simply trying all possible partition sizes and monitoring the resulting application performance. Another solution is to take a more analytical approach by using the cache miss rate curve of the application to determine an appropriate size. Such a curve, as shown in Figure 1.4, reveals the trade-off spectrum between provisioned cache size and the resulting cache miss rate. It characterizes the cache space requirements of an application. Obtaining the miss rate curve of the on-chip cache in an online manner is a challenging problem on current processors, and it remains to be seen whether future processors will provide the necessary additional hardware support.

In Figure 1.4, the cache miss rate curve for an artificial application indicates that provisioning just 5% of the cache results in a miss rate of 80%, but that minor increases in the amount cache would have no impact until at least 25% of the cache is provisioned. Once past the 25% mark, there is a drastic benefit from increasing this amount until 45% of the cache is provisioned, at which point the cache miss rate drops from 80% to 30%. Provisioning any more than 45% of the cache has diminishing returns, decreasing the cache miss rate from 30% to 15%. Knowing this non-linear trade-off curve for each application provides powerful information for cache provisioning among multiple applications.

In Chapter 5, we develop and demonstrate a methodology to perform automated online cache provisioning. We demonstrate how the hardware performance monitoring unit (PMU) and its associated hardware performance counters found in current commodity processors can be used to obtain a low-overhead, online approximation of the miss rate curve of the on-chip shared cache. On average, it requires a single probing period of 147 ms and subsequently 83 ms to process the data. We show the accuracy of this approximation and its effectiveness when applied to the shared cache management principle of providing isolation so that an appropriate amount of cache space can be allocated to an application. Using this methodology, we demonstrate performance improvements of up to 27% in terms of instructions-per-cycle (IPC).

1.3 Collaboration with Colleagues

Although the work in this thesis dissertation involved technical discussions and collaboration with colleagues, the core ideas, design, implementation, experimentation, and results were conceived, executed, and obtained by me.

In Chapter 3 on promoting sharing, I conceived the core idea of using PMUs to observe cross-chip traffic and appropriately cluster threads. I designed and implemented the thread migration component and the workload configurations to exhibit the desired workload characteristics, as well as conducted the experiments. Expertise in hardware PMUs was provided by Reza Azimi, a fellow graduate student. We jointly designed the memory region tracking component and the thread clustering algorithm. The RUBiS database workload was generously provided by Professor Cristiana Amza and Gokul Soundararajan. This work resulted in a publication in the 2007 European Conference on Computer Systems (EuroSys), titled, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors" [Tam et al. 2007b]. This publication forms the basis of Chapter 3, and has been cited by papers at major conferences, workshops, and magazines, such as ASPLOS [Eyerman and Eeckhout 2010; Gummaraju et al. 2008], SOSP [Baumann et al. 2009], OSDI [Boyd-Wickizer et al. 2008], HotOS [Boyd-Wickizer et al. 2009], PPoPP [Zhang et al. 2010], Communications of the ACM [Fedorova et al. 2010], and IEEE Micro Magazine [Knauerhase et al. 2008][4]. Other citations include: [Bhadauria et al. 2008; Blagodurov et al. 2009; Blelloch et al. 2008; Durbhakula 2008; Hakeem et al. 2009; Jiang et al. 2010ab; Sáez et al. 2008; Schneider 2009; Shen and Jiang 2009; Sondag and Rajan 2009; Srikantaiah et al. 2009a; Sudheer et al. 2008; Vaddagiri et al. 2009; Xian et al. 2008].

In Chapter 4 on providing isolation, I conceived the idea of a software-based cache partitioning mechanism in my 2004 thesis proposal. I designed and implemented both the mechanism and the workload configurations, as well as conducted all experiments. Livio Soares, a fellow graduate student, provided additional performance analysis of the data in Section 4.6.2. This work resulted in a publication in the 2007 Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), titled, "Managing Shared L2 Caches on Multicore Systems in Software" [Tam et al. 2007a]. This publication forms the basis of Chapter 4, and has been cited by papers at major conferences, such as ASPLOS [Shen 2010; Srikantaiah et al. 2008], ISCA [Hardavellas et al. 2009], EuroSys [Zhang et al. 2009a], USENIX Annual Technical Conference [Zhang et al. 2009b], HPCA [Awasthi et al. 2009; Lin et al. 2008], PACT [Fedorova et al. 2007], and PPoPP [Hofmeyr et al. 2010]. Other citations include: [Chan 2009; Guan et al. 2009; Jin et al. 2009; Kotera 2009; Kumar and Delgrande 2009; Liu et al. 2009; Raj et al. 2009; Wu and Yeung 2009].

In Chapter 5 on provisioning the shared cache, I pursued the idea to track every L2 cache

[4] © ACM, 2007. Chapter 3 is a minor revision of the work published in Proceedings of the 2^{nd} ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (March 21–23, 2007), http://doi.acm.org/10.1145/1272996.1273004

access using PMUs in order to generate approximated L2 cache miss rate curves (MRCs) online. I designed and implemented the RapidMRC system, as well as conducted the experiments. Reza provided the initial template code to configure the PMU. Well-known optimizations to Mattson's stack algorithm were implemented by Livio. This work resulted in a publication in the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), titled, "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations" [Tam et al. 2009]. This publication forms the basis of Chapter 5, and has been cited by papers at major conferences and magazines, such as ASPLOS [Zhuravlev et al. 2010], MICRO [Chaudhuri 2009], ISPASS [Xu et al. 2010], and Communications of the ACM [Fedorova et al. 2009 2010][5]. Other citations include: [Blagodurov et al. 2009; Rai et al. 2009; Walsh 2009].

1.4 Research Contributions

The contributions that stem from our research are demonstrated by the three publications generated from this thesis. To the best of our knowledge, we are the first, on commodity multicore systems, to develop and demonstrate a methodology for the operating system to:

1. promote sharing by clustering sharing threads based on runtime information obtained using hardware performance monitoring units (PMUs) [Tam et al. 2007b];
2. provide isolation by controlling occupation in the shared cache [Tam et al. 2007a];
3. provision the shared cache by approximating L2 cache miss rate curves online using hardware PMUs [Tam et al. 2009].

In each case, we developed a new mechanism and provided experimental evidence that it can be used to achieve performance gains. In two of our contributions, we also specifically contribute to the hardware PMU research community by concretely demonstrating profitable online usage cases of specific hardware PMU features [Azimi et al. 2009]. The three mechanisms developed as a part of our work create further research possibilities. Fellow researchers can use these initial mechanisms and ideas as a foundation or starting point for their own work, either further using, improving, and extending the base mechanisms, or to spawn or inspire related ideas. Chapter 7 describes some of the possible future work enabled by this dissertation. We hope we have opened up new research possibilities to the research community.

[5] © ACM, 2009. Chapter 5 is a minor revision of the work published in Proceedings of the 14^{th} International Conference on Architectural Support for Programming Languages and Operating Systems 2009 (March 7–11, 2009), http://doi.acm.org/10.1145/1508244.1508259

Chapter 2

Background

"Those who cannot learn from history are doomed to repeat it." – George Santayana

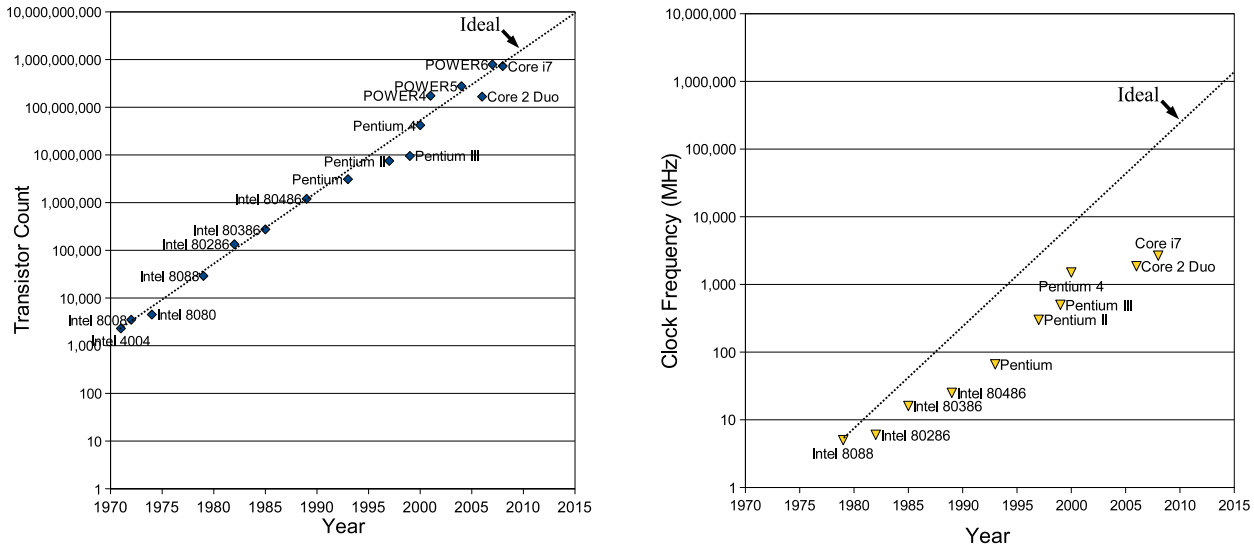
The purpose of this chapter is to provide a perspective on how this dissertation fits into the context of computer systems development, reviewing relevant advances in order to understand where we came from, how we arrived at the current state, and where we are potentially headed towards in the near and long term future. Subsequent chapters will each contain a section on prior work directly related to that chapter.

In this chapter, we first describe the evolution of hardware, followed by the evolution of operating systems. We describe the recurring pattern of how operating systems have been modified to first make functional usage of and then subsequently exploit new hardware features. We see how operating systems have actively managed shared, finite hardware resources to achieve performance improvements. Our research follows this well-accepted pattern of development and thus contributes to the evolution of computer systems.

2.1 Hardware Evolution

In this section, we review the evolution of relevant hardware, beginning with the introduction of processors that fit onto a single integrated circuit, also known as microprocessors. From these processors, we describe how multiple-core processors came into existence, why they have become prevalent, and why we believe that they are here to stay. Understanding the evolution of hardware enables software researchers to investigate changes in the operating system to address current and anticipated future hardware.

We describe the development of various forms of parallelism to extract performance gains from applications, such as pipeline, instruction-level, multiprocessor, and thread-level. These previous developments ultimately led to the development of multiple-core processors.



a. Moore's Law on transistor growth.

b. Impact on processor clock frequency.

Figure 2.1: Moore's Law on transistor growth and its impact on processor clock frequency [IBM 2007; Intel 2008a 2009 2005a; Sinharoy et al. 2005; Tendler et al. 2002]. Its impact on clock frequency has failed to keep up with the ideal trend, as shown for the Intel x86 family of processors.

2.1.1 Moore's Law is Important

The first commercially available microprocessor on an integrated circuit was the Intel 4004, released in 1971 [Faggin et al. 1996]. In 1975, Gordon Moore famously predicted that the number of transistors on an integrated circuit would double every two years, revising his earlier prediction in 1965 that the number of transistors would double every year [Moore 1965 1975]. This prediction, known as Moore's Law, has been maintained, as shown in Figure 2.1a, because of advances in integrated circuit manufacturing that have continually shrunk the size of circuit elements.

At the transistor level, smaller circuit elements enable processors to operate at higher clock frequencies because the smaller transistors can be switched between the *on* and *off* states more quickly. In addition, signal propagation times are shorter because of shorter wires between these transistors.

The benefits from Moore's Law, faster transistors and an abundance of them enabling microarchitectural design improvements, have contributed to higher processor clock frequencies, as shown in Figure 2.1b for the Intel x86 processor family, although these increases have failed to keep up with the ideal trend. Thus, Moore's Law on transistor-count growth, although it does not make any prediction about processor clock frequency, has been very loosely correlated with such increases. Next, we review microarchitectural design improvements that have enabled further performance gains.

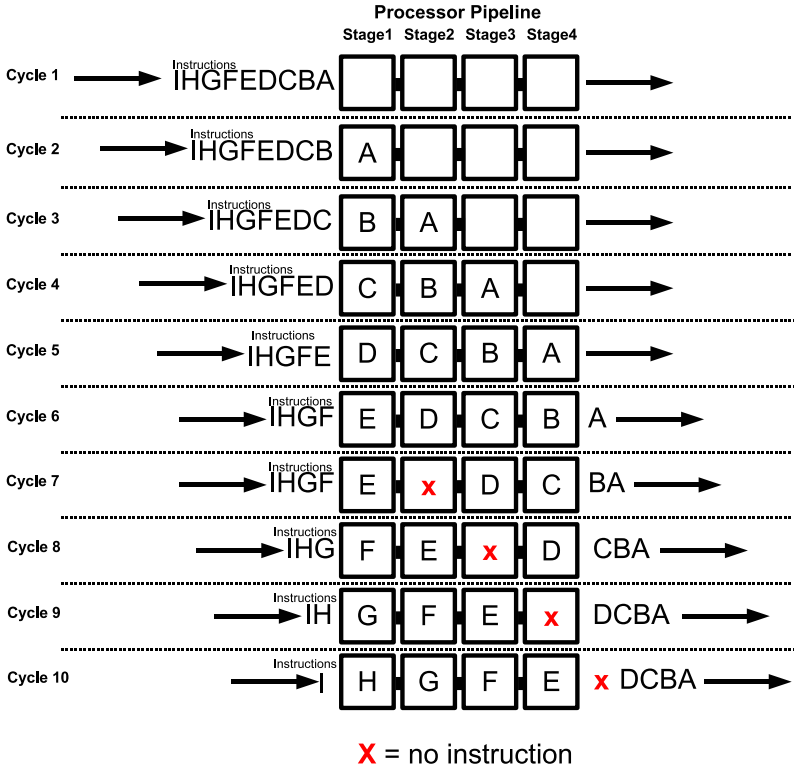


Figure 2.2: Parallelism by instruction pipelining. Although this technique enables higher processor clock frequencies, there may be cycles where no instructions are completed. Time progression is shown from top to bottom, while instructions flow through the processor pipeline from left to right. Each stage performs operations on a different instruction. A stall at clock cycle 7 in stage 2 eventually leads to no instruction completion at clock cycle 10.

Pipeline Parallelism

At the digital logic block level, another contributor to higher clock frequencies, given the abundance of transistors, is the use of the microprocessor architecture (microarchitecture) design technique of instruction pipelining. In pipelining, signal propagation through one large digital logic block is divided into several smaller digital logic block stages that have shorter signal propagation times, enabling shorter clock periods (higher clock frequencies). From a microarchitectural design perspective, instruction pipelining can be viewed as a form of parallelism because, at any point in time, when the pipeline is filled, each stage is performing an operation for a different instruction, such as shown in Figure 2.2 at clock cycle 6. In the figure, time progression is shown from top to bottom, while instructions flow through the processor pipeline from left to right. This instruction pipeline is analogous to the automobile assembly line where, at any point in time, each worker is operating in parallel to install a different component onto a different car, and by the end of the assembly line, the automobile is complete.

While the processor can operate at higher clock frequencies, the rate at which instructions are completed may not necessarily match. There may be clock cycles where no instructions are completed, as shown in Figure 2.2 at clock cycle 10, meaning that the average instructions-completed

per cycle (IPC) may fall below the ideal value of 1. This failure is due to stalls in the pipeline, which can come from a variety sources, including unavailable inputs to a pipeline stage, an operation within a stage taking several cycles to complete, or the inability of a pipeline stage to offload its outputs because of a full queue. For example, data may not be present because it is located off-chip in main memory, or floating-point operations may take several cycles to complete. In Figure 2.2, clock cycle 7 shows an example of where there was a stall in stage 2 of the pipeline that eventually leads to the lack of instruction completion at clock cycle 10.

In an attempt to reduce pipeline stalls and restore the IPC value to 1, various microarchitectural features were added to the pipeline, such as write forwarding, out-of-order execution, register renaming, and speculative execution based on branch prediction [Hennessy and Patterson 2007]. To deal with off-chip latencies leading to pipeline stalls, additional on-chip features have been included, such as on-chip floating-point units, various levels of caches, cache prefetching, and on-chip memory controllers. Moore's Law on transistor-count growth enabled all of these microarchitectural additions to fit within the processor chip.

Instruction-Level Parallelism

In an attempt to further raise the IPC value of the processor pipeline, microarchitecture researchers have added features to widen the pipeline, enabling potentially more than one instruction to complete within a single clock cycle, which is referred to as superscalar processing [Wall 1993]. Conceptually, these features attempt to extract parallelism from the application program, which is represented by a serial stream of instructions. They attempt to take the serial stream of instructions and extract instructions that can be executed in parallel with each other to shorten the total execution time, as shown in Figure 2.3. It is important to note that only instructions that do not directly or indirectly depend upon each other for input or output values can be executed in parallel. Given this constraint, the number of instructions that can be executed in parallel is known as the available instruction-level parallelism (ILP) of the serial instruction stream. For example, in Figure 2.3, instructions `B` and `C` depend only upon the result of instruction `A`. Instruction `D` depends only upon the result of instruction `B`. Instruction `E` depends only upon the result of instruction `C`. The resulting ILP of the instruction stream is 2 at its maximum and $1.\bar{6}$ on average. To exploit the available ILP, some features were introduced to widen the pipeline, including: multiple instruction issue and execution, multiple functional units, and vector units [Hennessy and Patterson 2007]. Again, Moore's Law on transistor-count growth enabled these microarchitectural additions to fit within the processor chip.

These hardware improvements, combined with compiler optimizations to exploit these facilities, were conceived with the goal of requiring no application-level modifications. Thus, software applications have had the luxury of experiencing performance gains without requiring application redesign or re-implementation.

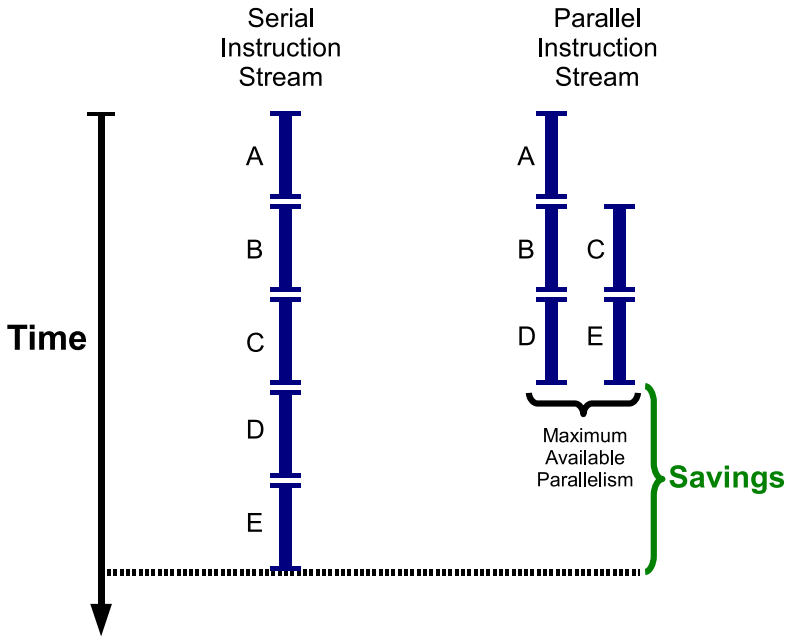


Figure 2.3: To increase the number of instructions completed per processor clock cycle, parallel instructions can be extracted from a serial instruction stream as long as they adhere to data dependencies between instructions. If instructions `B` and `C` depend only upon the result of instruction `A`, instruction `D` depends only upon the result of instruction `B`, and instruction `E` depends only upon the result of instruction `C`, then the maximum available instruction-level parallelism for this instruction stream is 2.

Multiprocessor Parallelism

Despite these performance improvements to single processors, one can always find larger and more sophisticated problems that need to be solved more quickly. A single processor still cannot meet the performance requirements of numerous important applications. This deficit led to the application of multiple microprocessors, within one computer and across multiple computers, leading to increased computer systems research in the area known as parallel and distributed systems. Examples of these multiple processor systems include SMP (symmetric multiprocessing) multiprocessors; NUMA (non-uniform memory access latency) multiprocessors; and clusters consisting of multiple computers (workstations, servers, blades), each one potentially being a multiprocessor itself.

Unfortunately, these multiple processor systems have largely remained in a specialized, niche computing domain which has had little applicability to the majority of the general computer world. It remains debatable as to the reason for the unpopularity of these traditional multiple processor systems. There appears to be many factors, such as (1) the difficulty in developing parallel versions of applications since software modifications are required, unlike the previous hardware improvements, (2) the once seemingly never-ending exponential single processor performance improvements incidentally linked to Moore's Law on transistor counts, and (3) the nonlinear hardware costs associated with increasing multiprocessor system size. Perhaps it is the combined effect of these factors that have allowed application developers to make more productive use of their time by simply waiting a few months for faster processors to become available for the same price as the previously acquired

ones, enabling single-processor applications to execute faster without additional programmer effort. However, this phenomenon has largely come to an end because processor manufacturers are now unable to viably develop processors with higher clock frequencies.

2.1.2 Moore's Law is Impotent

The end to single processor speed increases has little to do with Moore's Law. Moore's Law makes no claim with respect to processor speeds, but only on the number of transistors on a chip. Due to various electrical engineering factors, such as power dissipation and the constant speed of light affecting signal propagation delays, the frequency at which these transistors can be switched *on* and *off* is reaching its limit, which leads to a limit on the processor clock frequency. In addition, there are microarchitectural limitations, such as the complexity of extracting additional pipeline parallelism and instruction-level parallelism (ILP) from the serial instruction execution stream, the increasing complexity of monolithic processor designs, and relatively longer on-chip wire delays in traversing the chip [Wall 1993]. Straight-forward extensions to existing monolithic microarchitectural innovations, outlined in the previous subsections, appear to be inadequate for extracting further performance improvements from the executing software instructions.

From a mathematical perspective, performance within a processor family, such as the Intel x86 family, can be measured by the number of instructions completed per second (IPS: instructions per second), and is a result of the following.

$$IPS\ \frac{\text{instructions}}{\text{second}} \quad = \quad IPC\ \frac{\text{instructions}}{\text{cycle}} \quad \times \quad clock\_frequency\ \frac{\text{cycles}}{\text{second}} \tag{2.1}$$

To increase the value of IPS, and thus performance, $clock\_frequency$ could be increased, or IPC could be increased. Unfortunately, $clock\_frequency$ now appears capped and IPC has limited potential due to difficulty in extracting instruction-level-parallelism (ILP) from a single serial instruction stream. Moore's Law no longer correlates to higher $clock\_frequency$ values. However, Moore's Law does provide extra transistors that can help increase IPC in other ways.

Thread-Level Parallelism

From an architectural perspective, now that the easily-available instruction-level parallelism (ILP) has been harvested, the next unexploited source for IPC improvements lies in the availability of thread-level parallelism (TLP). TLP describes the scenario where there are multiple, independent threads of execution, which can be run simultaneously within a single processor. These multiple threads can come from either within a single application or across multiple applications. This scenario is similar to the traditional multiprocessor parallelism scenario described in a previous section, where multiple threads are executed on multiple processors. By exploiting thread-level parallelism, microarchitecture researchers have taken a page out of the book of computer systems researchers, inheriting some of the advantages and disadvantages in exploiting multiprocessor par-

allelism. Thread-level parallelism enables the IPC term in Equation 2.1 to become the sum of the IPC of each thread, as shown in Equation 2.2 for N threads.

$$IPS = \left(\sum_{n=1}^{N} IPC_n \right) \times clock\_frequency \qquad \text{, given } N \text{ threads} \tag{2.2}$$

The potential of thread-level parallelism within a single processor led to research in, and development of, various initial forms of hardware multithreading, and eventually led to the development and commercial availability of simultaneous multithreading (SMT) processors [Lo et al. 1997; Marr et al. 2002; Tullsen et al. 1995]. Similar to how a software-based multithreading mechanism maintains the context of multiple threads in main memory, SMT processors maintain the context of multiple threads but in designated hardware structures. With such hardware support, these threads are also known as *hardware threads*. This hardware extension enables the microarchitectural components to simultaneously process instructions from multiple hardware threads, rather than from just a single thread, thus extracting the parallelism found across multiple threads. In fact, hardware multithreading leverages much of the existing microarchitecture infrastructure that was previously designed to extract instruction-level parallelism (ILP) from a single thread, by feeding multiple independent streams of instructions into the processor pipeline [Burns and Gaudiot 2002]. SMT processors offer approximately a 25% performance increase on average, while costing only 5% more in transistor count, with a relatively straight-forward retrofit to existing out-of-order pipelined, superscalar microarchitectures [Koufaty and Marr 2003]. From a software point of view, for both the operating system and applications, SMT processors appear like multiple processors, with each hardware thread appearing as a separate traditional processor. This view enabled existing multiprocessor operating systems to run on SMT processors with few modifications, if any.

Memory Wall

Another issue limiting performance is the large and growing disparity between raw processor speeds and off-chip main memory speeds in terms of both latency and bandwidth. This disparity is commonly referred to as the *memory wall*. Access latencies to off-chip versus on-chip memory is typically an order of magnitude larger. For example, in the IBM POWER5 processor, accessing the on-chip L2 cache requires 14 processor cycles, whereas accessing the off-chip local main memory requires 280 cycles. In particular, the problem of limited off-chip memory bandwidth on processors has been noted by numerous researchers [Burger et al. 1996; Huh et al. 2001; Rogers et al. 2009]. While the number of transistors on a chip has increased at a rapid rate, the number of external pins and their associated speeds have not increased proportionately. Future processors will encounter the fundamental problem of shipping data into and out of the chip quickly enough, in terms of latency and bandwidth, to meet the demands of the ever-growing number of processing elements, caused by Moore's Law on transistor-count growth.

2.1.3 Moore's Law is Omnipotent: Rise of the (Multicore) Machines

The potential of thread-level parallelism (TLP), in combination with the unrelenting pace of Moore's Law, the uncoupling of processor speed from transistor count, and the rising memory wall, subsequently led to the development and commercial availability of multiple-core (multicore) processors. From a mathematical perspective, Equation 2.2 again meant that multiple threads, from TLP, was the only source of performance increase. In fact, the only feasible avenue for future performance increases appears, at this time, to be from increasing the number of hardware threads and from increasing the number of cores, given that the IPC of an individual thread appears to have reached its limit.

Multicore processors consist of several execution cores located on a single chip, in contrast to traditional single-core processors, which contain only a single execution core. SMT processors, in comparison, consist of a single core in which many microarchitectural resources within the core are shared by multiple hardware threads, whereas in multicore processors, microarchitectural resources within a core are private and only the on-chip L2 or L3 cache is potentially shared among the cores. SMT and multicore technologies are complementary and can be combined to create multicore-SMT processors, such as the IBM POWER5 or Intel Core 2 processors, both consisting of 2 cores in which each core contains 2 SMT threads. An illustration of the IBM POWER5 multicore-SMT processor is show in Figure 2.4a.

With transistor counts doubling periodically, chip designers could simply stamp out twice the number of cores on a chip and easily double the *raw* processing capabilities, assuming that there are enough threads to utilize all available cores. In effect, it can be seen as a divide and conquer approach. Now, the only problem left in this distributed environment on the chip, from a hardware point of view, is communication. Microarchitectural research can place more effort in on-chip interconnection networks, and on-chip cache organizations. Hierarchical, non-uniform designs appear to be the future of multicore processors and are now an active research area for the computer architecture community [Beckmann and Wood 2004; Dybdahl et al. 2007; Huh et al. 2005; Zhang and Asanović 2005].

Multiple-core processors were initially given the term, "single-chip multiprocessors", or more simply "chip multiprocessors" (CMP), because they appeared similar to a traditional SMP multiprocessor but with all of its processors located on a single chip. Eventually, once Intel finally began selling these kinds of processors, a simpler name evolved and they are now popularly known as "multicore" processors.

While the challenge of effectively utilizing the abundance of transistors, from a hardware perspective, has been addressed by the rise of multicore processors, the challenge of the memory wall still persists. Architecture and compiler researchers have explored solutions such as hardware-based [Chen and Baer 1995], compiler-extracted [Luk and Mowry 1999], and programmer-directed [Cher et al. 2004] prefetching. This line of research concentrates on the problems of predict-

ing what to prefetch, when to prefetch it so that it is available in a timely manner given the large memory wall, how to balance prefetcher memory requests against program-requested (demand-based) memory requests, and how to prevent prefetched data from polluting the demand-based data in the on-chip caches. The scope of prefetching research is limited to predicting what data will be required in the future and making it available in a timely manner. It does not deal with the management of demand-based data, such as promoting shared use of demand-based data among multiple threads, and preventing interference among demand-based data among multiple applications. Prefetcher research is complementary to operating system runtime management of on-chip shared caches.

As in the past, people will always find larger problems to solve that will exceed the performance capacity of a single processor system, even if it is a multicore processor. Thus, similar to the past, computer systems consisting of several chips, this time, each being a multicore processor, will serve an important role in fulfilling these higher-end computing needs. Major computer systems vendors, such as IBM, HP, Dell, and Sun, currently sell these multichip systems, which are more commonly referred to as *multisocket* systems because each chip sits in its own socket on the system board. Multisocket system boards for AMD and Intel multicore processors are also commercially available for user-assembled systems. As in the past, in order to fully utilize the full performance potential of this hardware, the operating system must be aware of these new hardware characteristics and optimize for them.

2.1.4 Multicore Processor Architecture Research

Architecture researchers at Stanford University were among the first to advocate general-purpose multicore processors [Olukotun et al. 1996]. Sohi went further and advocated radical re-architecting of multicore chips [Sohi 2003]. There have been several research multicore processors, such as Hydra [Hammond et al. 2000], Piranha [Barroso et al. 2000], and Atlas [Codrescu et al. 2001]. These projects explored issues such as microarchitectural design, compiler support, and speculative execution of user-level applications. Operating system design has not been a major focus of these projects.

Hardware-oriented research in on-chip cache organizations of multicore processors has been an active area because these caches are critical to overcoming the memory wall between processor and main memory speeds. Architecture researchers have proposed variations to: (1) the cache coherence protocols of various cache organizations [Chang and Sohi 2007 2006; Chishti et al. 2005; Liu et al. 2004]; (2) the cache line insertion [Jaleel et al. 2008] policy; (3) the cache line replacement policy [Dybdahl et al. 2006; Qureshi and Patt 2006; Srikantaiah et al. 2008; Suh et al. 2004]; (4) the sizes of caches and interconnects [Hsu et al. 2005; Zhao et al. 2007b]; (5) the organization of the caches and interconnects, such as tiled, and non-uniform hierarchies [Dybdahl et al. 2007; Huh et al. 2005; Kumar et al. 2005]; and (6) policies affecting performance, fairness, or quality of service [Hsu et al. 2005 2006; Iyer 2004; Iyer et al. 2007; Kannan et al. 2006; Zhou et al. 2009]. Again, operating

system design or co-design has not been a major focus of these projects.

The focus of this dissertation is not on microarchitectural improvements, but on how to exploit the given new hardware characteristics of multicore processors from an operating system's point of view, extracting as much performance as possible. While architecture researchers play an important role in finding good and efficient uses of more and more transistors, operating systems researchers have the potentially multiple research roles of (1) exploiting the given hardware architectures to extract the most performance for the software levels above, and (2) making suggestions to architecture researchers on how the interaction between the architecture and operating system can be improved to extract more performance from hardware. In this dissertation, we concentrate on the first role, but also provide a few suggestions on additional hardware support.

2.1.5 Multicore Processor Architectures Today

Multicore processors have existed in the realm of embedded devices since at least 1994. Some examples of such embedded multicore processors include: (1) the Texas Instruments TMS320C80 multimedia video processor in 1994, consisting of 1 general-purpose core and 4 digital signal processing cores [Texas Instruments 1994]; (2) the Motorola MPC821 PowerQUICC communications processor in 1995, consisting of 1 general-purpose core and 1 special-purpose communications core [Motorola 1998]; (3) the Broadcom BCM1250 network processor in 2001, consisting of 2 general-purpose cores [Broadcom 2001]; and (4) the Sun MAJC 5200 media processor, announced in 1999 and finally shipped in 2002, consisting of 2 general-purpose cores [Sun 1999 2002].

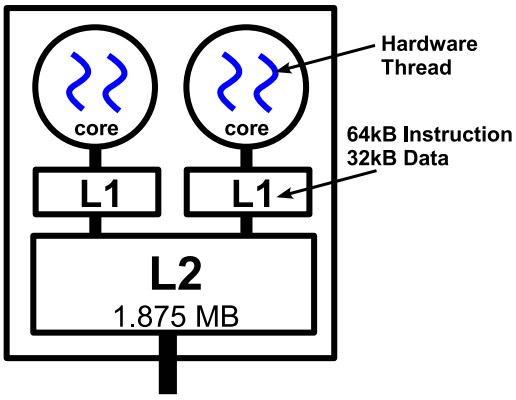
In the realm of general-purpose computing (i.e., laptop, desktop, and server computers), general-purpose multicore processors have been commercially available since 2001 with the release of the IBM POWER4 multicore processor. Table 2.1 shows a time line of commercially available multicore processors for general-purpose computers. It indicates wide gaps in time between the introduction of the first general-purpose multicore processor by IBM in 2001, the subsequent releases by other general-purpose processor manufacturers in 2004, and the eventual mainstream acceptance of general-purpose multicore processors in 2005, signalled by the release of multicore processors by Intel and AMD. Our work in this dissertation is applicable to both embedded and general-purpose multicore processors that contain on-chip shared caches.

A common design theme is the use of on-chip shared caches, typically at the final cache level before requiring access to off-chip caches or main memory, commonly known as the *last-level* cache. For most processors, this last-level cache is the L2 cache, and for a few others, this last-level cache is the L3 cache. These two characteristics are specified in the columns labelled "Shared Caches" and "On-Chip Caches" in Table 2.1, respectively.

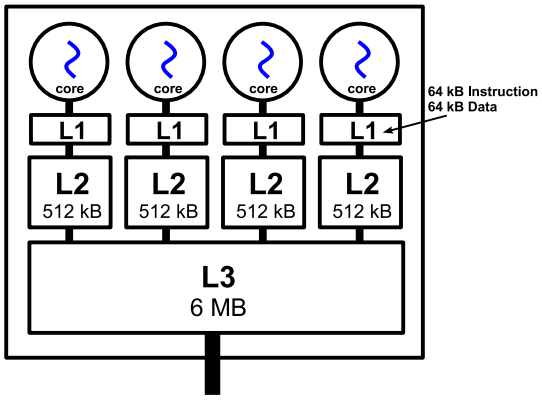
As concrete examples of the high-level organization of the multiple cores and caches within these commercially available processors, Figure 2.4 shows simplified diagrams of a 2-core IBM POWER5, a 4-core AMD "Shanghai" Opteron, and an 8-core Sun "Niagara 2" UltraSPARC T2 processor. The POWER5 contains a 1.875 MB L2 cache that is shared among 4 SMT hardware threads from

Ship Date	Multicore Processor	On-Chip Caches	Shared Caches	Citations
2001				
Oct.	2-core IBM POWER4	L1,L2	L2	[IBM 2001]
2002				
2003				
2004				
Feb.	2-core HP PA-8800	L1	none	[HP 2004]
Mar.	2-core Sun UltraSPARC IV	L1	none	[Sun 2004ab]
May	2-core IBM POWER5	L1,L2	L2	[IBM 2004]
2005				
Apr.	2-core Intel Pentium	L1,L2	none	[Intel 2005b]
Apr.	2-core AMD Opteron	L1,L2	none	[AMD 2005]
Oct.	2-core IBM PowerPC970MP	L1,L2	none	[Apple 2005; IBM 2005b]
Nov.	3-core IBM-Microsoft PowerPC Xenon	L1,L2	L2	[Microsoft 2005]
Dec.	8-core Sun UltraSPARC T1	L1,L2	L2	[Sun 2005]
2006				
Jul.	2-core Intel Itanium 2	L1,L2,L3	none	[Intel 2006b]
Sept.	9-core Sony-Toshiba-IBM Cell	L1,L2	none	[IBM 2006; Sony 2006]
Nov.	4-core Intel Xeon & Core 2 Extreme	L1,L2	L2	[Intel 2006a]
2007				
Apr.	2-core Sun-Fujitsu SPARC64 VI	L1,L2	L2	[Fujitsu 2007]
May	2-core IBM POWER6	L1,L2	none	[IBM 2007]
Sept.	4-core AMD Opteron	L1,L2,L3	L3	[AMD 2007]
Oct.	8-core Sun UltraSPARC T2	L1,L2	L2	[Sun 2007]
2008				
Sept.	6-core Intel Xeon	L1,L2,L3	L3	[Intel 2008b]
Oct.	4-core Sun-Fujitsu SPARC64 VII	L1,L2	L2	[Fujitsu 2008]
2009				
Jun.	6-core AMD Opteron	L1,L2,L3	L3	[AMD 2009]
2010				
Feb.	4-core Intel Itanium 2	L1,L2,L3	none	[Intel 2010a]
Feb.	8-core IBM POWER7	L1,L2,L3	L3	[IBM 2010]
Mar.	8-core Intel Xeon	L1,L2,L3	L3	[Intel 2010b]

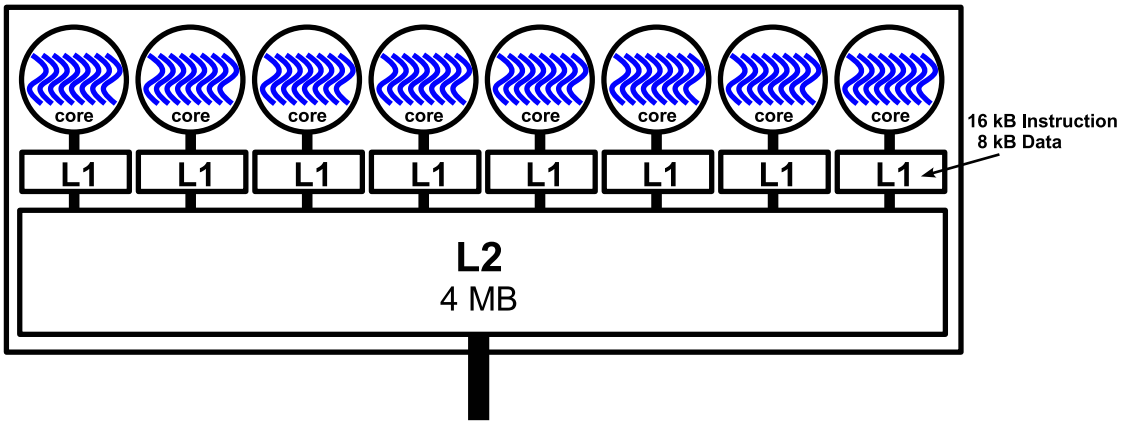
Table 2.1: Time line of commercially available multicore processors for general-purpose computers (i.e., laptops, desktops, and servers). This time line indicates wide gaps in time between the introduction of the first general-purpose multicore processor by IBM in 2001, the subsequent releases by other general-purpose processor manufacturers in 2004, and the eventual mainstream acceptance of general-purpose multicore processors in 2005, signalled by the release of multicore processors by Intel and AMD. For each processor, the depth of the on-chip memory hierarchy and the caches that are shared among cores are also specified.



a. 2-core IBM POWER5



b. 4-core AMD "Shanghai" Opteron



c. 8-core Sun UltraSPARC T2

Figure 2.4: High-level internal organization of commercially available multicore processors. A common characteristic of these multicore processors is that they share the last-level cache among all cores.

2 cores; the "Shanghai" Opteron contains a 6 MB L3 cache that is shared among 4 cores; and the "Niagara 2" UltraSPARC T2 contains a 4 MB L2 cache that is shared among 64 hardware threads from 8 cores.

Shared caches have become a popular design choice in addressing the memory wall problem. On-chip caches have always been a limited resource with major performance implications, since accessing off-chip caches or main memory typically takes an order of magnitude longer than on-chip cache accesses. Shared caches have the advantage of maximizing the amount of on-chip cache space available to all cores and hardware threads, thereby reducing the frequency of off-chip accesses. The main performance trade-offs, willingly accepted by the designers, are higher access latency to the shared cache, compared to a private cache, and the potential for cache space interference among cores or hardware threads. This dissertation demonstrates, in Chapter 4, how the operating system can manage the shared cache to prevent cache space interference among the cores.

2.1.6 Key Hardware Characteristic of On-Chip Shared Caches

On multicore processors, sharing of on-chip caches is very fine-grained in terms of time, because at every processor cycle, it is possible for several cores to attempt to access the shared cache simultaneously. As described by Parekh et al. and Lo, these processors[1] can suffer from fine-grain conflicts in the caches at the granularity of a cycle [Lo 1998; Parekh et al. 2000]. This type of cache behaviour on multicore processors may lead to severe performance degradations [Chandra et al. 2005; Fedorova et al. 2005; Lin et al. 2008; Snavely and Tullsen 2000; Tam et al. 2007a].

In contrast, traditional single-core processor caches are primarily considered to be private (even though they are shared among multiple applications) because they are accessed exclusively by only one application for the duration of its processor time-slice. The relatively long time-slices, in the order of millions of processor cycles, allow an application to warm up the cache at the start of each time-slice, by initially bringing in content from main memory, and then exploit the warmed cache, by reusing the content, for a long period of time before reaching the end of the time-slice.

This dissertation demonstrates how the operating system can manage the on-chip shared cache, given its fundamental hardware characteristic of fine-grained sharing. With this characteristic in mind, we maximize the potential for fast sharing among cores of the chip by *promoting sharing*, and minimize the potential for cache space interference among cores by *providing isolation*. In other words, we strive to maximize the performance advantage and minimize the performance disadvantage of the on-chip shared cache.

2.2 Operating System Evolution

In this section, we describe the evolution of operating systems in response to the underlying hardware evolution. The recurring pattern that will be described is that operating systems evolve first to make functional usage of a new hardware feature, and then to exploit that new feature for performance improvements. The operating system affects application performance primarily through scheduling and memory management, which are fundamental tasks of modern operating systems. With each new hardware feature, changes were made to memory management or scheduling algorithms in order to optimize for the given hardware feature. This pattern of evolution is also followed by our research.

The need for operating system design to account for hardware characteristics has been advocated by many researchers. Perhaps the most well-known advocate is Ousterhout, who, in 1990, examined the failure of systems software performance improvements to track hardware performance improvements [Ousterhout 1990]. Other researchers include Anderson et al., Chen and Bershad, Liedtke, Rosenblum et al., and Torrellas et al.. Anderson et al., in 1991, described how operating system researchers had failed to account for the hardware characteristics of RISC (reduced

[1] They describe SMT processors, but their observation applies equally to multicore processors.

instruction set computer) processors, compared to CISC (complex instruction set computer) processors, resulting in operating system performance improvements that were well below application performance improvements [Anderson et al. 1991]. As well, Anderson et al. also described how microarchitecture researchers had failed to account for the needs of operating systems when they designed RISC processors. Chen and Bershad, in 1993, re-evaluated commonly held assertions about operating system behaviour and its impact on the memory hierarchy [Chen and Bershad 1993]. In turn, Liedtke, in 1995, further analyzed the data of Chen and Bershad, drawing additional conclusions [Liedtke 1995]. Rosenblum et al., in 1995, examined how the introduction of out-of-order superscalar processors and other hardware improvements would affect operating system performance [Rosenblum et al. 1995]. Torrellas et al., in 1998, examined how the operating system could be modified to expose cache locality patterns to the underlying hardware [Torrellas et al. 1998].

2.2.1 Single Processor Optimizations

In the context of single processor computer systems, the addition of new hardware features to the microprocessor architecture, such as a hardware memory management unit (MMU), a very large memory address space, and an on-chip L2 cache have caused operating systems to evolve to make use of and to exploit these new hardware features.

Originally, the MMU was introduced to allow operating systems to provide memory protection between multiple running applications, greatly increasing computer system stability. After making the necessary modifications in the operating system, other features were added to operating systems to exploit the MMU for performance gains by further evolving the virtual memory management system.

MMU: Performance Optimizations

One performance optimization enabled by the MMU was the ability to perform memory-mapped file I/O (input/output), which reduced the number of memory copies traditionally required to perform these operations [Fitzgerald and Rashid 1986; Rashid et al. 1988]. In addition to reducing the number of memory copies, memory-mapped file I/O can reduce the total number of I/O operations because only the accessed pages of the file-mapped region are transferred to/from disk, rather than all data of a designated buffer.

Another performance optimization enabled by the MMU was the use of copy-on-write (COW) pages, which reduced the number of pages copied upon application startup, thereby increasing application startup speed [Rashid et al. 1988]. The MMU enabled the operating system to share existing physical pages between applications and create a separate copy only when an application attempts to modify (write to) the shared page.

Another type of functional capability enabled by the MMU and its resulting virtual memory

mechanism was the ability to run large applications that would have otherwise not fit in the available physical memory, or to run a larger collection of smaller applications. The virtual memory mechanism, combined with memory pages transferring to/from swap space located on a hard disk, enabled this capability. Physical memory pages were treated as a cache to disk swap space. After making the necessary modifications to the operating system to provide this feature, performance improvements to this new level of the memory hierarchy (disk) were incorporated.

Performance improvements were required because of the large speed disparity between main memory and disk. Operating system researchers explored two forms of speculation in attempts to hide the speed disparities: (1) predicting which page in main memory is the least likely to be re-accessed so as to evict it to swap space during page replacement; (2) predicting which page in swap space is most likely to be accessed soon so as to page it into main memory. Investigations into the first form of speculation led to various page replacement algorithms [Belady 1966; Carr and Hennessy 1981; Corbato 1968; Jiang et al. 2005]. Investigations into the second form of speculation led to various prefetching algorithms, also known as anticipatory paging, or swap prefetching, or prepaging [Black et al. 1991; Kaplan et al. 2002]. In addition to swap space management, eviction and prefetching have also been explored within the context of the file system buffer cache management, used to cache blocks belonging to files [Jiang and Zhang 2002; McKusick et al. 1984; Patterson et al. 1995; Zhou et al. 2004]. These examples demonstrate how the operating system can improve application performance by intelligently managing the finite hardware resource of physical memory pages, shared among multiple applications.

The MMU enabled operating systems to provide virtual memory addressing to applications. Each virtual memory page is mapped to a physical memory page using a page table stored in main memory. To improve the speed of address translation from virtual to physical page address, a small on-chip hardware cache, known as the translation look-aside buffer (TLB), was added to the processor to temporarily hold page table entries. Operating systems were modified mainly to appropriately synchronize page table entries located in main memory and TLB entries located on-chip. Performance optimizations were explored to reduce pressure on this small, finite TLB resource. Moreover, given that some MMUs support multiple physical page sizes, researchers have examined modifying the operating system to support larger page sizes in an attempt to increase address space coverage of the TLB, thereby reducing TLB miss rates which lead to application performance improvements [Cascaval et al. 2006; Navarro et al. 2002; Romer et al. 1995]. Their work are examples of how the operating system can intelligently manage the finite hardware resource of TLB entries to improve application performance.

64-bit Memory Address Space: Performance Optimizations

The MMU enabled memory protection between multiple running applications by allowing each application to exist in its own, protected, virtual memory address space. However, there is a performance cost to co-operating applications that frequently share data across memory address

spaces. With the commercial availability of processors with an extremely large, 64-bit memory address space, researchers, such as Chase et al. and Heiser et al., have explored the performance benefits of single-address-space operating systems, in contrast to traditional operating systems that feature multiple address spaces [Chase et al. 1994; Heiser et al. 1998]. Single-address-space operating systems, while maintaining adequate memory protection between applications, eliminate this need to cross memory address spaces, since there is only a single memory address space which is shared by all applications and the operating system.

On-Chip Cache: Performance Optimizations

With the introduction of on-chip L2 and L3 caches, operating system researchers examined ways to increase hit rates in these types of caches. One technique examined is called cohort scheduling, where upon context switching to another application, the operating system selects the application with the greatest chance of finding its content in the cache, rather than simply picking the next application in the run queue [Larus and Parkes 2002]. Bellosa proposed using TLB information obtained from hardware performance monitoring units (PMUs) to achieve this goal [Bellosa 1997]. Along similar lines but in a more forceful way, increasing hit rates in the on-chip L1 instruction cache was studied by Harizopoulos and Ailamaki [Harizopoulos and Ailamaki 2004]. They demonstrated that in a multithreaded database server application, context switching between threads can be done at a finer time granularity and with careful selection of the next thread so that the L1 instruction cache content has maximum reuse across all suitable threads. These optimizations demonstrate how the operating system can manage the hardware resource of the on-chip L1 and L2 caches for application performance improvements.

In real-time operating systems, Liedtke et al. demonstrated how to partition the on-chip L2 cache to give a real-time application an isolated region of the cache in order to maintain performance guarantees [Liedtke et al. 1997]. They modified the memory management system to make use of page-coloring in order to implement cache partitioning on commodity processors. They were the first to implement software-based cache partitioning in the operating system. This page-coloring technique is also used in our research, described in Chapter 4 in the context of multicore processors that contain an on-chip shared L2 cache.

2.2.2 Exploiting Multiple Processors

Relative to single processors, there are many more operating system adaptations and performance optimizations required for multiprocessors in order to handle real parallel activities as opposed to pseudo-concurrency, via time-slicing, on single processor systems. There are multiple processors, caches, main memory banks, interconnects, and I/O devices that must be kept busy, co-ordinated, and evenly load-balanced. The operating system is responsible for the intelligent management of these shared hardware resources. These hardware performance issues are typically addressed by

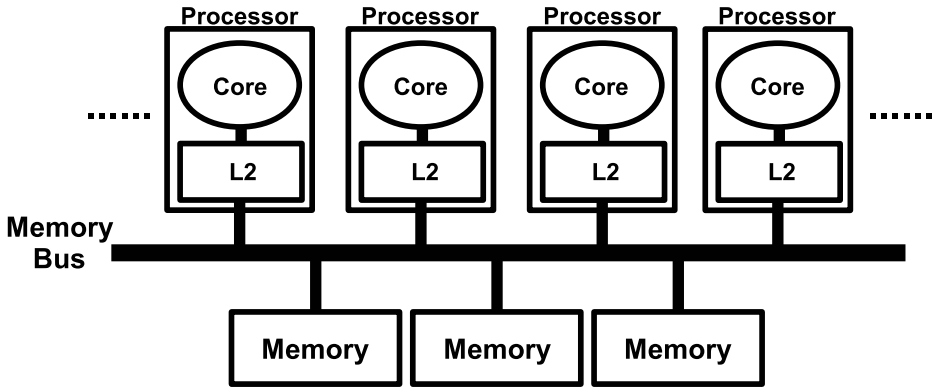


Figure 2.5: A simplified view of an SMP multiprocessor. The SMP multiprocessor consists of several unicore processors connected by a single shared memory bus.

the operating system via scheduling and memory management.

Mukherjee et al. provide a survey of performance optimizations that have been applied to multiprocessor operating systems [Mukherjee et al. 1993]. Torrellas et al. characterized the caching characteristics of multiprocessor operating systems [Torrellas et al. 1992]. Gamsa et al. demonstrated the performance advantages in fundamentally designing an operating system for multiprocessors, focusing on the scalability principles of maximizing concurrency and maximizing locality [Gamsa 1999; Gamsa et al. 1999]. Xia and Torrellas examined operating system techniques as well as additional hardware features that can help improve multiprocessor operating system performance [Xia and Torrellas 1999].

SMP Multiprocessors

SMP (symmetric multiprocessing) multiprocessors have existed since at least 1962, with the release of the Burroughs D825 Modular Data Processing System computer and its accompanying operating system – the Automatic Operating and Scheduling Program [Anderson et al. 1962; Enslow 1977; Thompson and Wilkinson 1963]. In general, operating systems were first modified to function with multiple processors and then optimized for performance gains. With SMP multiprocessors, as shown in Figure 2.5, the main hardware characteristics to be considered for performance maximization by the operating system are the utilization of the processors, the on-chip L2 cache of each processor, and the bandwidth of the single shared memory bus. The performance optimizations fall under two broad principles of (1) maximizing concurrency to make full use of all available processors, and (2) maximizing locality to make maximum use of the local on-chip L2 cache of each processor. These are the two principles of performance scalability described by Gamsa [Gamsa 1999].

Operating systems evolved to enable more and more concurrency within the kernel in order to exploit all of the available processors. Often, single processor operating systems were retrofitted to enable them to function on multiprocessors. Modifying traditional single processor Unix to execute correctly and efficiently was described by various researchers [Bach and Buroff 1984; Janssens et al.

1986; Russell and Waterman 1987]. The natural course of retrofitting was to first allow only a single designated *master* processor to execute operating system kernel code, leaving the remaining *slave* processors to execute user-level applications [Enslow 1977; Goble and Marsh 1982]. To mildly improve performance, any processor was allowed to execute kernel code, by placing a single, coarse-grained lock around the entire kernel to allow only one processor to be in the kernel at any time. These modifications were among the simplest solutions to ensure functional correctness of the operating system, however, at the expense of performance. As more parallel performance was demanded from the operating system, multiple and finer-grained locks were used to allow multiple processors to execute kernel code simultaneously [Enslow 1977; Kleiman et al. 1992; Peacock et al. 1992]. These optimizations belong to the first category of maximizing concurrency and can be viewed as the operating system managing and maximizing the use of the finite hardware resource of processors.

As an example of this optimization trend of maximizing concurrency, the initial use of a single job queue, shared among all processors, for scheduling was described by researchers such as Enslow, and Denham et al. [Denham et al. 1994; Enslow 1977]. This single job queue originated from a single-processor operating system, and by surrounding it with a lock, was made operable on multiprocessors. In such a design, only one processor can perform scheduling operations at any point in time since the globally shared data structures are protected by a single lock. Depending on hardware characteristics of the multiprocessor, such as the inter-processor interrupt latency and L1 cache line transfer latency, a different design may be more efficient. The scheduler could be modified from a single, globally shared queue to a local queue for each processor, each with its own lock. Depending on the hardware characteristics, this design can greatly reducing lock contention on the queues, enabling all processors to perform scheduling simultaneously [Denham et al. 1994; Enslow 1977; Ni and Wu 1989].

With the use of multiple job queues, load imbalance can occur, where one processor becomes idle because there are no jobs to run from its local queue, despite available jobs located in other queues. Ironically, the original single job queue could not suffer from this problem. Consequently, operating systems also perform load balancing among the multiple queues by moving tasks between the queues. However, aggressive load balancing can cause higher than normal cache miss rates upon context switching to a freshly migrated task. In contrast, a non-migrated task may find some of its content in the L2 cache upon context switching in for execution. Thus, cache affinity considerations were also taken into account when deciding whether to move a task to another queue [Gupta et al. 1991; Squillante and Lazowska 1993; Torrellas et al. 1993; Vaswani and Zahorjan 1991]. This example shows a scenario where the optimization goal of maximizing concurrency is moderated by the goal of maximizing locality. Cache affinity consideration can be viewed as the operating system managing the use of the finite hardware resource of on-chip L2 cache space.

With the heavy use of fine-grained locks to increase concurrency, the locks themselves became a performance bottleneck. Performance optimization of locks were examined by many researchers.

The trade-offs between spinning versus blocking versus spin-then-block locks were explored by researchers [Anderson 1990; Gupta et al. 1991; Mukherjee and Schwan 1993]. To prevent ping-ponging of the cache line containing the lock variable between processors due to cache coherence enforcement by the hardware, Mellor-Crummey and Scott examined methods to mitigate the problem [Mellor-Crummey and Scott 1991]. This optimization can be seen as maximizing the locality of the lock meta-data structures. To further increase concurrency and match the available hardware parallelism, the use of read-writer locks was first examined by Courtois et al., allowing for either multiple simultaneous readers or a single writer to acquire the lock of a globally shared data structure [Courtois et al. 1971].

False sharing of global kernel data structures can cause performance problems in multiprocessors. For example, a 128-byte cache line may contain data structure `A` in the first 64 bytes and data structure `B` in the last 64 bytes. If one processor repeatedly accesses data structure `A` (reads and writes) while another processor repeatedly accesses data structure `B` (reads and writes), the underlying cache line is considered shared by the hardware, requiring hardware cache coherence actions, greatly reducing performance. Consequently, researchers examined the benefits of increasing the locality of global data structures via cache line padding [Denham et al. 1994; Gamsa et al. 1999; Tam 2003]. Cache line padding prevents disparate data structures from occupying the same cache line and being falsely shared. For example, with cache line padding, data structure `A` would solely occupy one cache line and data structure `B` would solely occupy another cache line. This kind of memory allocation minimizes cache line sharing, since two different data structures occupy separate and distinct cache lines. The cache lines remains local to their processor, thus temporarily experiencing increased locality.

Saturation of the globally shared memory bus was a problem examined by Antonopoulos et al. [Antonopoulos et al. 2003]. Scheduler modifications were made to manage bus bandwidth by co-scheduling applications that together would not saturate the bus with their memory accesses. Their work demonstrates an example of how the operating system can intelligently manage the shared finite hardware resource of bus bandwidth in order to improve application performance.

Although the iterative approach of decomposing coarse-grained locks on existing globally shared kernel data structures into multiple finer-grained locks is one approach to extracting performance from multiprocessors, this approach has limitations when scaling to a large number of processors [Appavoo 2005; Gamsa 1999]. The approach focuses mainly on maximizing concurrency, to make maximum use of all available processors, without directly addressing locality issues, which becomes an important factor in large multiprocessor systems. Consequently, researchers have explored the performance scalability benefits of eliminating all globally shared data structures by using fully distributed, local data structures and code paths, developing a scalable multiprocessor operating system from scratch rather than retrofitting an existing single processor operating system [Gamsa et al. 1999]. The systematic use of distributed, local kernel data structures and code paths not only eliminated many lock requirements and resulting lock contention, but eliminated

many bad caching effects that would have been problematic in large scale multiprocessors due to the amount of cache coherence traffic required. It enabled maximum use of each local on-chip L2 cache of each processor, reducing communication across the global bus for the purposes of accessing main memory or remote L2 caches. Their research demonstrates how operating systems can be designed to optimally use the available finite hardware resources, enabling performance scalability on multiprocessors.

NUMA Multiprocessors

Hardware scalability limitations of single bus SMP multiprocessors led to NUMA (non-uniform memory access latency) multiprocessors, which are arranged in a hierarchical manner. They may consist of several SMP multiprocessors (nodes) with a fast interconnection network of memory buses, as shown in Figure 2.6. In NUMA multiprocessors, all memory is accessible by all processors and a hardware cache coherence protocol is used to keep all L2 caches appropriately synchronized. With this hardware support for shared memory, NUMA multiprocessors typically run a single operating system that spans all nodes. The main hardware considerations for the operating system are the non-uniform access latency to local versus remote memory, and the non-uniform access latency to L2 caches located in local versus remote nodes. The distributed nature of these machines means that the operating system must focus on maximizing locality [Appavoo et al. 2007; Gamsa 1999]. A beneficial side effect of maximizing locality is that it helps to increase concurrency due to fewer global synchronization requirements.

Multicore processor systems share some general hardware characteristics with NUMA multiprocessor systems. Multicore systems have a shared memory hierarchy, each level with a different latency, which is similar to NUMA systems. However, this hierarchy can exist within a single chip and between multiple multicore chips. NUMA systems today, such as the SGI Altix system [SGI 2006], are composed of multicore chips rather than traditional unicore chips.

A NUMA operating system must take into account this non-uniform latency when managing memory. NUMA memory management research has examined page placement, migration, and replication to improve memory locality. Memory page placement deals with selecting a physical page from a memory bank location to maximize local memory usage [Ho 2004; LaRowe and Ellis 1991; Marathe and Mueller 2006]. Memory page migration deals with subsequently moving the page contents to a different physical page that offers better local memory usage [Bolosky et al. 1989; Corbalan et al. 2003; LaRowe et al. 1991; Tikir and Hollingsworth 2004; Verghese et al. 1996; Wilson and Aglietti 2001]. Finally, memory replication deals with subsequently creating duplicate physical pages to allow multiple processors to each simultaneously experience higher local memory usage [Bolosky et al. 1989; Chapin et al. 1995; Verghese et al. 1996].

A NUMA operating system must also take into account the non-uniform memory latency when scheduling jobs. NUMA scheduling research has examined thread migration to maintain and improve locality. When performing load balancing among the multiple run queues, memory locality

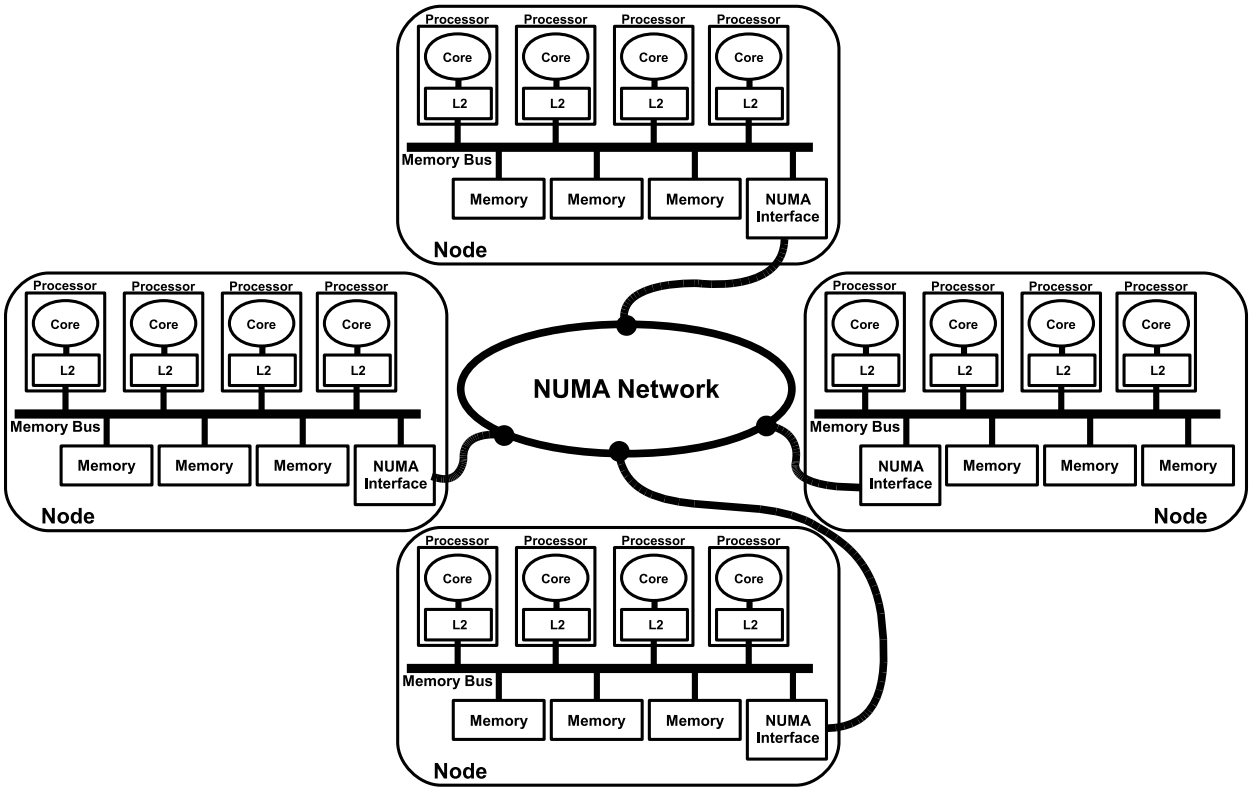


Figure 2.6: A simplified view of a NUMA multiprocessor. The NUMA multiprocessor consists of several SMP multiprocessors (nodes) with a fast interconnection network of memory buses.

must be considered [Aas 2005; Bellosa and Steckermeier 1996; Zhou and Brecht 1991]. A hierarchical load-balancing method was examined and implemented in the Linux operating system, where migrating jobs between processors within a node to balance load is attempted before migrating jobs between processors across nodes [Aas 2005]. Jobs running on one node will very likely have memory content located within local memory, while migrating the job across nodes will result in remote memory accesses to its former node. Although page migration could be invoked to move appropriate content to the new local memory, the overhead of page copying would need to be considered. To further improve memory locality, Bellosa and Steckermeier examined how to detect sharing among threads scattered across the various nodes and subsequently schedule them onto the same node to increase memory locality [Bellosa and Steckermeier 1996].

In our work on thread clustering in Chapter 3, we attempt to maximize locality of the application running on a multicore multiprocessor system using an approach that is similar to Bellosa and Steckermeier [Bellosa and Steckermeier 1996]. Bellosa and Steckermeier were unable to successfully achieve performance improvements using hardware performance monitoring unit (PMU) technology of the early 1990s, while we were successful in our implementation using newer hardware PMU technology. Although our technique was examined on multi-chipped multicore processors, it can also be applied to NUMA processors. We detect sharing among threads and migrate them to a single multicore chip to encourage shared use of the on-chip shared L2 cache. Analogously, our

technique could be applied to detect sharing among threads scattered across various nodes of a NUMA multiprocessor system, and migrate them to a single node to encourage shared use of the local memory of the node.

At the University of Toronto, the Hurricane and Tornado research operating systems focused on performance scalability of the operating system on large NUMA multiprocessor systems [Gamsa et al. 1999; Unrau et al. 1995]. They examined how to build an operating system from the ground up for performance scalability. The Tornado operating system demonstrated the two performance scalability principles of maximizing concurrency and maximizing locality. Multicore processors are a natural extension of the NUMA hardware platform in which performance scalability principles equally apply.

There has been a vast amount of research done on improving software performance in the realm of NUMA systems research. The main lesson that we can learn from past NUMA systems research is that because of the latencies in the memory hierarchy, maximizing locality is very important, and this lesson is equally applicable to multicore systems.

2.2.3 Exploiting Simultaneous Multithreading

Unlike traditional processors, in simultaneous multithreading (SMT) processors, many microarchitectural hardware resources are shared among multiple threads of execution, leading to potential interference between threads. In addition to the processor pipeline resources, on-chip caches are also shared, which includes the L1 instruction, L1 data, and L2 caches. The operating system must consider how to manage these shared hardware resources, via memory management and scheduling, in order to maximize application performance.

To operating systems, SMT processors appear as a traditional multiprocessor, enabling standard SMP multiprocessor operating systems to function on these system. An initial examination of operating system behaviour on SMT processors was conducted by Redstone et al. [Redstone et al. 2000].

The first performance problem addressed was busy waits that consumed too many microarchitectural resources, causing significant interference with other threads [Redstone 2002]. SMT-aware operating systems, on Intel SMT processors, have been modified to prevent over-consumption of shared microarchitectural resources by: (1) using the `halt` instruction rather than busy-spinning in a idle loop when the processor becomes idle, and (2) using the `pause` instruction rather than busy-spinning in a spinlock loop [Microsoft 2002; Nakajima and Pallipadi 2002].

The operating system can manage SMT shared hardware resources at a coarse granularity through careful co-scheduling of tasks. That is, it can select tasks to run simultaneously, each running in an SMT hardware thread. Careful co-scheduling can reduce contention and improve application performance, as initially shown by Snavely and Tullsen [Snavely and Tullsen 2000], and subsequently by many others [Bulpin and Pratt 2005; Fedorova et al. 2004 2005 2006; McDowell et al. 2003; Nakajima and Pallipadi 2002; Parekh et al. 2000; Snavely et al. 2002]. Snavely and

Tullsen were concerned with the interference impact on the shared on-chip caches, while Nakajima and Pallipadi were more concerned with the interference impact on various functional units of the processor pipeline [Nakajima and Pallipadi 2002]. The co-scheduling goals of minimizing misses in the shared cache are equally applicable to multicore processors.

Fedorova et al. demonstrated the technique of compensative scheduling, modifying the scheduler to give larger or smaller time slices to applications to compensate for interference to an application [Fedorova et al. 2007]. For processors that offer thread priority adjustability, such as the IBM POWER5, this lever can also be used to compensate for interference in the processor pipeline, as shown by Meswani and Teller [Meswani and Teller 2006]. These scheduling approaches of compensating for interference among multiple applications are equally applicable to multicore processors. Duty-cycle modulation may also be a possible technique for compensating for interference, as investigated by Zhang et al. [Zhang et al. 2009b].

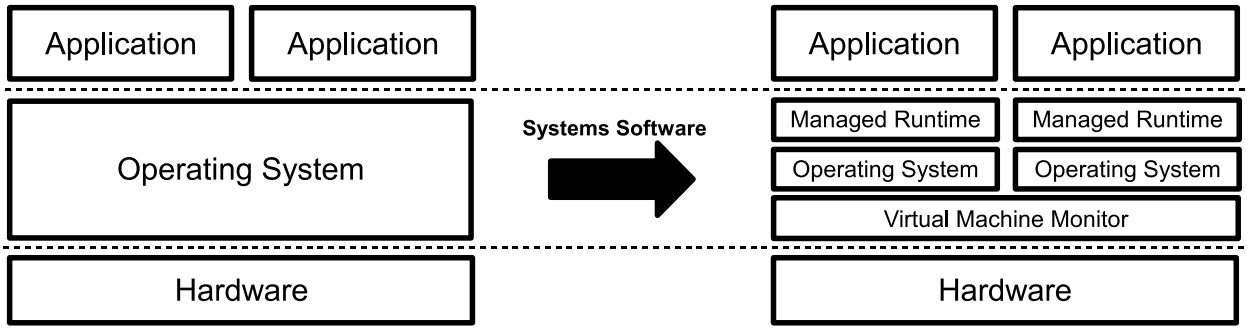
Partitioning the shared L1 cache in SMT processors, via compiler-based application code transformations, was examined by Nikolopoulos [Nikolopoulos 2004]. Partitioning the shared L2 cache in SMT processors, with the aid of hardware extensions was first examined by Suh et al. [Suh et al. 2001ab 2002 2004]. Our work in Chapter 4 demonstrates how to provide this isolation in the shared L2 cache of multicore processors using a software-only cache partitioning mechanism, without any additional hardware support or explicit application transformations.

2.2.4 Exploiting Multiple Cores

On multicore processors, the main hardware property that must be considered by the operating system is that there can be on-chip caches shared by multiple cores. Typically, this includes the L2 cache or, if it is present, the L3 cache. In contrast, the L1 caches are private to each core, unlike in SMT processors. Another hardware property that should be considered by the operating system is that communication among cores is faster than on traditional multi-chipped multiprocessors because all cores are located on the same chip, sharing the same on-chip L2 cache.

From past operating system developments described in the previous sections, we can see that operating system management of the on-chip shared caches, a new shared hardware resource, is a natural evolution. There is currently no explicit shared-cache management by the operating system in current systems.

In 2004, Fedorova et al. first described the need for operating systems to account for the hardware characteristics of multicore and hardware multithreaded processors [Fedorova et al. 2004]. Since then, numerous researchers have investigated how to improve multicore performance, as will be described in the related work sections of the next three chapters. In terms of advocating a fundamental redesign of the operating system for multicore processors rather than simpler retrofitting, such investigations first began in 2007 with the McRT project [Saha et al. 2007], followed by Barrelfish/Multikernel [Baumann et al. 2009; Schüpbach et al. 2008], Corey [Boyd-Wickizer et al. 2008], Tessellation [Liu et al. 2009], and fos [Wentzlaff and Agarwal 2009]. These project are similar in



a. traditional system stack b. new system stack

Figure 2.7: Traditional and new system stacks. The once monolithic operating system layer has been decomposed into several layers.

spirit to the Hurricane and Tornado operating system fundamental redesign efforts from the NUMA multiprocessor era [Gamsa et al. 1999; Unrau et al. 1995]. Our work in this dissertation adds to these foundations by investigating small components that are critical to performance, and serve as a building block for performance scalability.

In this dissertation, we make operating system changes to the scheduler and memory management system in order to improve application performance. In Chapter 3, we modify the scheduler, co-scheduling threads to promote shared use of the shared L2 caches, thereby increasing locality and taking advantage of the faster communication between cores located on the same chip. In Chapter 4 we modify the memory management system, explicitly managing the shared cache to provide cache space isolation among applications.

2.2.5 Operating Systems: On the Move

The traditional responsibilities of the operating system are being gradually transferred to both the layer above and the layer below the system software stack, as shown in Figure 2.7. The once monolithic operating system layer is being decomposed into several layers of abstraction in an attempt to use a *divide and conquer* approach to handle system software complexity problems of today. The responsibility of scheduling multiple applications is effectively being given to the virtual machine monitor [Smith and Nair 2005], an optional layer below the operating system that gives the illusion of a dedicated stand-alone computer, thus allowing multiple operating systems to concurrently run on the system. The responsibility of managing memory is increasing being given to the managed runtime layer above the operating system.

In terms of the research community, MIT first examined the idea of a bare-bones, skeletal operating kernel, called Exokernel, combined with a user-level library, called LibOS, to provide traditional operating system runtime services [Engler et al. 1995]. In terms of Figure 2.7, Exokernel is the virtual machine monitor while LibOS is an application running along side other applications. More recently, the Denali isolation kernel followed this philosophy but specifically examined how to

provide scalability to potentially thousands of light-weight *guest* operating systems running Internet server applications [Whitaker et al. 2002]. In terms of Figure 2.7, Denali is the virtual machine monitor while multiple guest operating systems occupy the operating system layer. Along similar lines, the IBM Libra project examined the performance potential of placing a Java virtual machine runtime environment, in combination with a thin execution environment, directly on top of a virtual machine monitor rather than on top of a traditional operating system, given that thread scheduling and some memory management tasks are already performed by the runtime environment [Ammons et al. 2007]. In terms of Figure 2.7, the Java runtime is the operating system and the managed runtime layers, while the virtual machine monitor used is the Xen Hypervisor.

Specifically targetting multicore processors, the Intel McRT project focused on performance scalability of multicore processors by investigating a fundamental redesign of the entire system software stack, from virtual machine monitor, including the operating system, up to the managed runtime environment [Saha et al. 2007].

Despite the thinning out of the traditional operating system, no matter where parts of the traditional operating system may end up, or what it will be called (perhaps *systems software* is a better general term), the responsibilities and issues that it deals with will not disappear, only hidden temporarily or transformed into a different, perhaps slightly easier context in which to attempt to solve the problems. These system software issues must be dealt with, at whichever layer they end up.

On the Move Down to Virtual Machine Monitors

> *"Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem." – David Wheeler*

Perhaps due to the failure of operating system researchers, operating systems are perceived as a problem in which virtual machine monitors, an optional layer of indirection underneath, are seen as the practical solution. The problems that operating systems have failed to solve but which are seen to be easier to solve at the software layer below include: security, isolation, trust, reliability, fault-tolerance, compatibility, legacy preservation, extensibility, performance, dynamic machine migration, operating system design complexity, administration complexity, installation complexity, upgrading and patching complexity, customization capabilities, and server consolidation.

Adding a layer of indirection allows for a *divide and conquer* approach to solving these problems. Adding this layer underneath the operating system creates a new and different context in which to attempt to solve these problems. This lower level of machine abstraction of instructions, registers, interrupts, physical memory pages, and devices, is in contrast to the higher-level, operating system abstraction of processes, threads, operating system calls, interrupt service routines, file systems, and device drivers.

Research into the concept of virtual machine monitors and their uses began at IBM in the 1960s with the M44/44X [Denning 1981] and CP/CMS [Meyer and Seawright 1970] research projects.

With CP/CMS, IBM developed and pioneered the commercial feasibility of virtual machine monitors on the IBM System/360 computers, running the CP (Control Program) virtual machine monitor, with multiple instances of the CMS (Cambridge Monitor System) simple, single-user interactive operating system. The CP/CMS system used virtual machine monitors as a simplified way to implement multi-user time-sharing capabilities for a computer [Creasy 1981; Varian 1989]. At the time, batch-processing operating systems were prevalent and the concept of an interactive, multi-user, time-sharing operating system was a relatively new idea. Rather than design and implement complex modifications to the operating system itself to support time-sharing, they explored using a virtual machine monitor to run multiple simple existing operating systems, one for each connected user, to achieve the effect of time-sharing. Using virtual machine monitors allowed for a divide and conquer approach to providing time-sharing capabilities.

Approximately 40 years later, virtual machine monitors are now experiencing a rebirth in popularity. The same basic conditions that originally made virtual machines popular are appearing once again, such as large, expensive, centralized computer centres with a tremendous number and variety of users and workloads, this time coming from the Internet rather than terminal connections. Virtual machine monitor facilities have remained in IBM mainframe computers ever since the IBM System/360, and they are now also in IBM POWER processor-based computers. VMware, Inc. is considered the dominant commercial vendor of virtual machine monitor software for Intel/AMD x86 processor-based computers. Other companies, such as Xen/Citrix, Microsoft, Parallels, Sun, and Oracle, have developed and advocated their own versions or variations of a virtual machine monitor.

In the context of a virtual machine monitor running on multicore systems, shared-cache management principles still apply. Enright Jerger et al. provide examples of how performance is affected in such computing environments [Enright Jerger et al. 2007]. This piece of systems software must intelligently manage the shared cache used by multiple operating systems that are running on top of the virtual machine monitor. In its simplest form, this change in context can be seen as a change in labels, where shared-cache management is applied at a coarser granularity, to entire operating systems rather than individual applications.

On the Move Up to Managed Runtime Environments

Managed runtime environments are a popular language platform on which to run applications, especially Internet and Web-based applications. Examples of managed runtime environments include, Java, Microsoft .Net Common Language Runtime, Microsoft Silverlight, Adobe Flash, Adobe Integrated Runtime, Perl, Python, Ruby, and JavaScript runtime environments. These environments offer advantages such as just-in-time compilation, dynamic optimization, runtime type-safety checking, garbage collection, standardized systems-like libraries, a standardized execution environment, and portability among different underlying system software and hardware layers. These managed runtime environments perform operating system-like activities such as scheduling and

memory management for the application threads. Thread management duties may include thread creation, scheduling, synchronization, and destruction. Memory management duties may include memory allocation and garbage collection for application threads.

Shared-cache management principles still apply in the managed runtime layer, especially if the runtime layer is the sole entity running on a multicore system. This layer of systems software can intelligently manage the shared cache used by multiple application threads running within the managed runtime environment. These application threads may be working co-operatively, such as in a parallel scientific application, or they may be competing threads, such as for each thread servicing an incoming request in a Java application server.

Databases can also be considered as managed runtime environments. They tend to manage their own system resources that they have pre-allocated in bulk from the operating system. Operating system-like activities that they perform include managing their large blocks of memory, performing raw disk block device I/O instead of using the provided file system, and performing application-specific buffer pool management. Shared-cache management principles can be applied to databases as well, such as the thread clustering technique in Chapter 3, or the cache partitioning technique in Chapter 4, to keep performance-critical data structures in the on-chip shared cache of a multicore processor.

Chapter 3

Promoting Sharing in the Shared Cache

"A candle loses nothing by lighting another candle." – James Keller

[1]The shared nature of on-chip, last-level caches is a property that can be exploited for performance gains. Data and instructions that are simultaneously accessed by multiple cores within a single chip in a shared manner can be quickly reached by all cores if they are located in the shared caches. This hardware performance characteristic leads to our first principle of promoting sharing in the shared cache. An operating system scheduler could select processes or threads that share data or instructions and co-schedule them to all run at the same time within the same multicore processor so that they can exploit sharing in the cache. In contrast, current operating system schedulers are not aware of these shared on-chip caches. Consequently, on a multicore system consisting of ***multiple chips***, current operating systems distribute threads across several processor chips in a way that can cause many unnecessary, long-latency cross-chip cache accesses. By promoting sharing, we can mitigate the amount of cross-chip traffic, and exploit a major advantage of on-chip shared caches, namely its fast on-chip sharing capabilities.

In this chapter, we demonstrate the application and effectiveness of the principle of promoting sharing. We target large multithreaded commercial workloads that execute in a single-programmed (single application) computing environment. On a small-scale multi-chip platform, consisting of an *8-way* IBM POWER5 system containing 2 chips × 2 cores per chip × 2 hardware threads per core, we reduce processor pipeline stalls caused by cross-chip cache accesses by up to 70%, resulting in performance improvements of up to 7%. On a larger-scale multi-chip platform, consisting of a *32-way* IBM POWER5+ system containing 8 chips × 2 cores per chip × 2 hardware threads per core, measurements indicate the potential for up to 14% performance improvement.

[1] This chapter is a minor revision of the work published in Proceedings of the 2^{nd} ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (March 21–23, 2007), http://doi.acm.org/10.1145/1272996.1273004

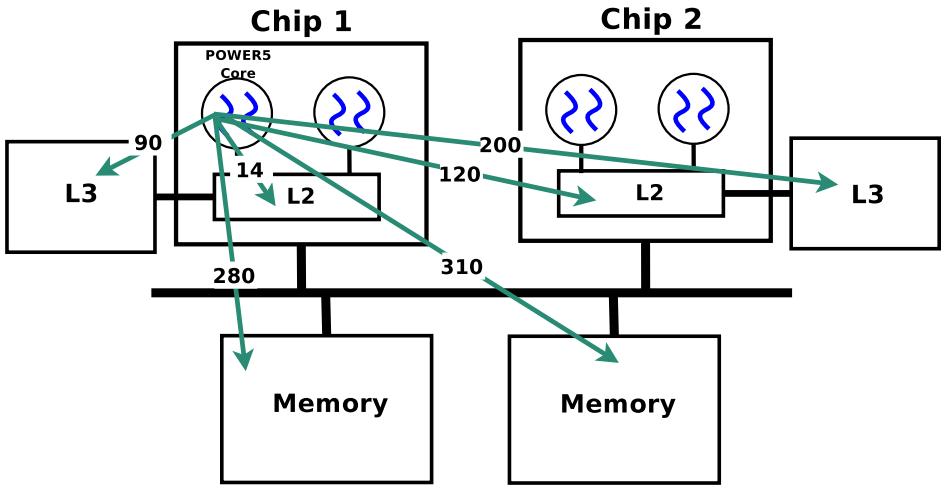


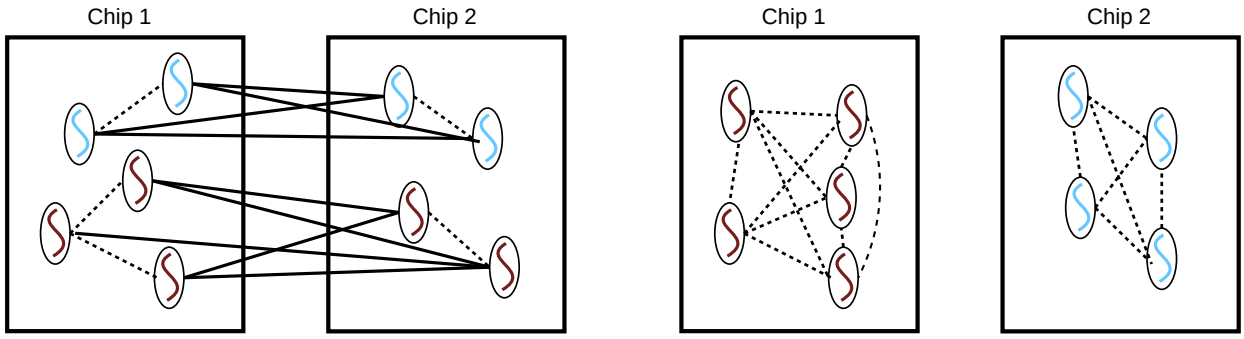
Figure 3.1: The IBM OpenPower 720 architecture. The numbers on the arrows indicate the access latency from a thread to different levels of the memory hierarchy, in terms of the number of processor cycles. Any cross-chip sharing takes at least 120 processor cycles.

We use operating system scheduling to promote shared use of the shared cache. We match the sharing that occurs in software to the available hardware sharing facilities.

3.1 Introduction

Shared memory multiprocessors of today consist of several chips, each of which is a multicore processor. A key difference between these newer multicore multiprocessor systems and traditional unicore multiprocessor systems is that the newer systems have non-uniform data sharing overheads. That is, the overhead of data sharing between two processing components differs depending on their physical location. As an example, consider the IBM OpenPower 720 latencies depicted in Figure 3.1. In these multichip multicore systems, for hardware threads that reside on the same core, communication typically occurs through a shared L1 cache, with a latency of 1 to 2 cycles. For hardware threads that do not reside on the same core but reside on the same chip, communication typically occurs through a shared L2 cache, with a latency of 10 to 20 cycles. Hardware threads that reside on separate chips communicate either through shared memory or through a cache-coherence protocol, both with an average latency of hundreds of cycles.

Current operating system schedulers on multichip multicore systems do not take the non-uniform sharing overheads into account. As a result, threads that share data heavily will not typically be co-located on the same chip. Figure 3.2 shows an example of a scenario where two clusters of threads are distributed across the processing units of two chips. The distribution is usually done as a result of some dynamic load-balancing scheme. If the volume of intra-cluster sharing is high, a default operating system scheduling algorithm (shown on the left) may result in many high-latency inter-chip communications (solid lines). If the operating system can promote sharing by detecting the thread sharing pattern and scheduling the threads accordingly (shown on



a. default b. clustered

Figure 3.2: Default versus clustered scheduling. The solid lines represent high-latency cross-chip communication, while the dashed lines represent low-latency intra-chip communication when sharing occurs within the on-chip L1 and L2 caches.

the right), then threads that communicate heavily could be scheduled to run on the same chip and, as a result, most of the communication (dashed lines) would take place in the form of on-chip L1 or L2 cache sharing.

Another benefit of promoting sharing by co-locating sharing threads onto the same chip is that they may incidentally perform prefetching of shared regions for each other. That is, they may help to obtain and maintain frequently used shared regions in the local cache.

Finally, threads that don't share data and have high memory footprints may be better placed onto different chips, helping to reduce potential cache capacity problems.

Detecting sharing patterns of threads automatically has been a challenge. One approach used in the past for implementing software distributed shared memory (DSM) exploited page protection mechanisms to identify active sharing among threads [Amza et al. 1996]. This approach has two serious drawbacks: (i) the page-level granularity of detecting sharing is relatively coarse with a high degree of false sharing, and (ii) the overhead of protecting pages results in high overhead with an attendant increase in page-table traversals and translation look-aside buffer (TLB) flushing operations.

In this chapter, we design, implement, and evaluate a scheme to schedule threads based on sharing patterns detected online using Azimi's thread sharing detection component [Azimi 2007]. This component detects sharing among threads with low overhead by using the data sampling features of the performance monitoring unit (PMU) available in today's processing units. The primary advantage of using the PMU infrastructure over page-level mechanisms is that the former is fine-grained, down to individual L2 cache lines, and has far lower overheads since most of the monitoring is offloaded to the hardware.

We have implemented this scheme in the Linux kernel running on an *8-way* IBM POWER5 SMP-CMP-SMT multiprocessor. For commercial-grade multithreaded server workloads (VolanoMark, SPECjbb, and RUBiS), we are able to demonstrate the benefits of promoting sharing, measured by significant reductions in processor pipeline stalls caused by cross-chip cache accesses of up to 70%. These reductions lead to performance improvements of up to 7%.

The specific workloads we target in our experiments are multithreaded commercial-grade server applications, such as databases, application servers, instant messaging servers, game servers, and mail servers. The programming model of these workloads is that there are multiple threads of execution, each handling a client request to completion. These threads of the application exhibit some degree of memory sharing, and thus make use of the shared memory programming paradigm, as opposed to message passing. The scheme we propose to promote sharing automatically detects clustered sharing patterns among these threads and groups these threads accordingly onto the processor chips.

In theory, promoting sharing by thread clustering may be done at the application level by the application programmer. However, it is fairly challenging for a programmer to determine the number of shared memory regions and the intensity of sharing between them statically at development time. Another problem with manual, application programmer-written thread clustering is the extra effort of re-inventing the wheel for every application. Additional complexities may arise when application code is composed from multiple sources, such as shared libraries, especially if the source code is not available. The dynamic nature of multiprogrammed computing environments is also difficult to account for during program development. Our automated scheme can detect sharing patterns that the application programmer may have been unaware of. In addition, our scheme can handle phase changes and automatically re-cluster threads accordingly.

As a motivational example, our scheme can be applied to the Java platform without requiring modifications to the application or managed runtime system. A Java application developer may write her multithreaded J2EE (Java 2 Platform, Enterprise Edition) servlet as usual and the underlying operating system would automatically promote sharing by detecting sharing among threads and clustering them accordingly.

3.2 Related Work

The work most closely related to our scheme of promoting sharing was done by Bellosa and Steckermeier [Bellosa and Steckermeier 1996]. They first suggested using hardware performance monitoring units (PMUs) and their associated hardware performance counters to detect sharing among threads and to co-locate them onto the same node of a NUMA multiprocessor. Due to the high costs of accessing hardware performance counters at the time, more than ten years ago on a Convex SPP 1000, they did not obtain publishable results for their implementation. The larger scope of their research focused on performance scalability of NUMA multiprocessors, stressing the importance of using locality information in thread scheduling.

Weissman proposed additional hardware PMU features to detect cache misses and reduce conflict and capacity misses [Weissman 1998]. Their system required user-level code annotations to manually and explicitly identify shared regions among threads in order to deal with sharing misses. Rajagopalan et al. also proposed a technique that required hints supplied by the programmer

about how to group threads [Rajagopalan et al. 2007]. In our work, we use Azimi's thread sharing detection component to automatically detect the shared regions in an online manner [Azimi 2007].

Thread clustering algorithms were examined by Thekkath and Eggers [Thekkath and Eggers 1994]. Their research dealt with finding the best way to group threads that share memory regions together onto the same processor so as to maximize cache sharing and reuse. However, they were not able to achieve performance improvements for the scientific workloads they used in their experiments. The two main factors cited were (1) the global sharing of many data structures, and (2) the fact that data sharing in these hand-optimized parallel programs often occurred in a sequential manner, one thread after another. In contrast, our chosen workloads (1) exhibit non-global, clustered sharing patterns and (2) are not hand-optimized multithreaded programs but are written as client-server applications that exhibit unstructured, intimate sharing of data regions. Their work focused on the clustering algorithm, assuming that the shared-region information is known a priori, and was evaluated in a simulator. In contrast, our work solves the missing link by using Azimi's thread sharing detection component to detect these shared regions in an online, low-overhead manner on real hardware running a real operating system [Azimi 2007]. Since our focus is not on the clustering algorithm itself, we used a relatively simple, low-overhead algorithm.

Sridharan et al. examined a technique to detect user-space lock sharing among multithreaded applications by annotating user-level synchronization libraries [Sridharan et al. 2006]. Using this information, threads sharing the same highly-contended lock are migrated onto the same processor. Our work adopts the same spirit but at a more general level that is applicable to any kind of memory region sharing. Locks could be considered a specific form of memory region sharing, where the region holds the lock mechanism. Consequently, our technique implicitly accounts for lock sharing among threads.

Bellosa proposed using TLB information to reduce cache misses across context switches and maximized cache reuse by identifying threads that share the same data regions [Bellosa 1997]. Threads that share regions are scheduled sequentially, one after each other so as to maximize the chance of cache reuse. Larus and Parkes had the same goals and explored a technique called cohort scheduling [Larus and Parkes 2002]. Koka and Lipasti also had the same goals and provided further cache miss details [Koka and Lipasti 2005]. The work of these three research groups was in the context of a uniprocessor system, in an attempt to maximize cache reuse of a single L2 cache, whereas our work targets multiple shared caches in a multicore multiprocessor system, in an attempt to maximize cache reuse within each shared cache.

Philbin et al. attempted to increase cache sharing reuse of single-threaded sequential programs by performing automatic parallelization, creating fine-grained threads that maximized cache reuse [Philbin et al. 1996].

In the realm of real-time systems, Anderson et al. investigated a scheduling algorithm that attempts to co-schedule threads belonging to the same application instead of co-scheduling threads belonging to different applications, to promote shared use of the shared cache, all while satisfying

real-time deadlines of each thread [Anderson et al. 2006].

In the domain of parallelizable computations, such as parallel sort algorithms, Chen et al. investigated fine-grained, application-specific scheduling techniques to promote constructive cache sharing on multicore processors [Chen et al. 2007]. The technique involves intelligently decomposing a large, parallelizable computation into threads, and controlling the order and location of thread execution. An automated compiler-based technique to achieve these goals was explored by Kandemir et al. [Kandemir et al. 2009].

In the realm of databases, Harizopoulos and Ailamaki explored a method to transparently, without application source code modifications, increase instruction cache sharing re-use by performing more frequent but intelligently chosen thread context switches [Harizopoulos and Ailamaki 2004]. Selecting threads belonging to the same stage may improve instruction cache reuse. The general staged-event driven architecture was described and explored by Welsh et al. [Welsh et al. 2001].

The remaining related work mostly concentrates on determining the best tasks to co-schedule in order to minimize capacity and conflict misses. Our work is targeted specifically at exploiting the shared aspect of shared caches in a multi-chip setting. Our work may be complementary to these past efforts in minimizing capacity and conflict misses of shared caches.

Many researchers have investigated minimizing cache conflict and capacity problems of shared L2 cache processors. Snavely and Tullsen did seminal work in the area of co-scheduling, demonstrating the problem of conventional scheduling and the potential performance benefits of symbiotic thread co-scheduling on a simulator platform [Snavely and Tullsen 2000]. With the arrival of Intel HyperThreaded multiprocessor systems, Nakajima and Pallipadi explored the impact of co-scheduling on these real systems [Nakajima and Pallipadi 2002]. Parekh et al. made use of hardware PMUs that provided cache miss information to perform smart co-scheduling [Parekh et al. 2000]. Others, such as McGregor et al. and El-Moursy et al., have found that on multiprocessors consisting of multiple SMT chips, cache interference alone was insufficient in determining the best co-schedules because SMT processors intimately share many microarchitectural resources in addition to the L1 and L2 caches [El-Moursy et al. 2006; McGregor et al. 2005]. McGregor et al. found that per-thread memory bandwidth utilization, bus transaction rate, and processor stall cycle rate were important factors. El-Moursy et al. found that the number of ready instructions and the number of in-flight instructions were important. Bulpin and Pratt also made use of hardware PMUs to derive a model for estimating symbiotic co-scheduling on an SMT processor [Bulpin and Pratt 2005]. Suh et al. described the general approach of memory-aware scheduling, where jobs were selected to run based on cache space consumption [Suh et al. 2001a 2002]. For example, a low cache consumption job was run in parallel with a high cache consumption job. Settle et al. proposed adding hardware activity vectors per cache line, creating a framework for exploring cache optimizations [Settle et al. 2004]. Their goal, within a single SMT chip context, was to minimize capacity and conflict misses. Fedorova et al. examined the issue of operating system scheduler redesign and explored co-scheduling to reduce cache conflict and capacity misses based on a model

of cache miss ratios [Fedorova et al. 2004 2005 2006].

3.3 Performance Monitoring Unit

Most modern microprocessors today have performance monitoring units (PMUs) with integrated hardware performance counters that can be used to monitor and analyze performance in real time. Hardware performance counters allow for the counting of detailed microarchitectural events in the processor, such as branch mispredictions and cache misses. They can be programmed to interrupt the processor when a specified number of specific events occur. Moreover, PMUs make various registers available for inspection, such as addresses that cause cache misses or the corresponding offending instructions.

However, PMUs in practice are difficult to use because (i) they can be extremely processor-specific, varying between generations within the same processor family, (ii) they lack documentation describing them in detail, (iii) there are a limited number of counters, and (iv) the counters have various constraints imposed upon them. For example, they do not provide enough counters to simultaneously monitor the many different types of events needed to form an overall understanding of performance. Moreover, hardware performance counters primarily count low-level microarchitectural events from which it is difficult to extract high-level insight required for identifying causes of performance problems.

We use fine-grained hardware performance counter multiplexing that was introduced by previous work to make a larger set of logical hardware performance counters available [Azimi et al. 2005]. This PMU infrastructure is also able to speculatively associate processor pipeline stalls to different causes. Figure 3.3 shows an example of stall breakdown for the VolanoMark application. The average cycles-per-instruction (CPI) of the application is divided into *completion cycles* and *stall cycles*. A completion cycle is a cycle in which at least one instruction is retired. A stall cycle is a cycle in which no instruction is completed, which can be due to a variety of reasons. Stalls are broken down based on their causes. With appropriate hardware configuration, stalls that are due to data cache misses can be further broken down according to the source from where the cache miss was satisfied. While it is possible to have a detailed breakdown of data cache misses according to their sources, for the purpose of this work, we are only interested in knowing whether the source was *local* or *remote*, where local refers to a cache on the same chip as the target thread, and remote refers to a cache on another chip. Although the L3 cache is often off-chip, we consider the L3 cache that is directly connected to a chip to be its local L3 cache.

3.4 Design of Automated Thread Clustering Mechanism

Our thread clustering approach consists of four phases.

1. **Monitoring Stall Breakdown:** Using hardware performance counters, processor pipeline

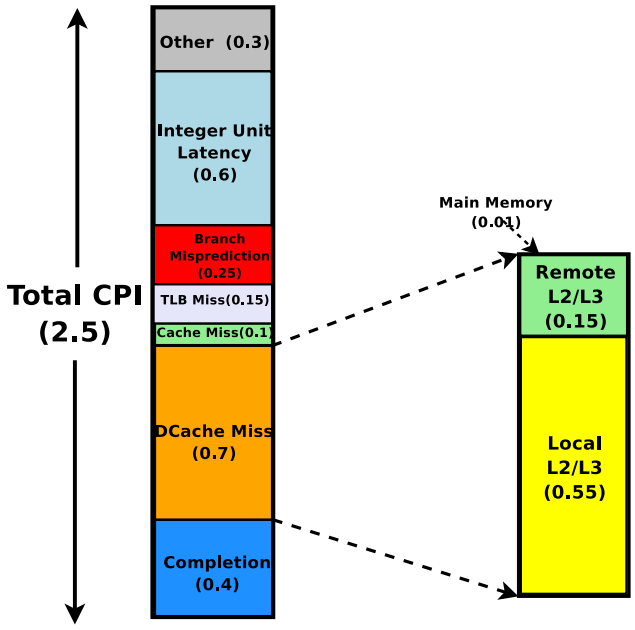


Figure 3.3: The stall breakdown for VolanoMark. The stalls due to data cache misses are further broken down according to the source from where the cache miss is eventually satisfied.

stall cycles are broken down and charged to different microprocessor components to determine whether cross-chip communication is performance limiting. If this is the case, then the second phase is entered.

2. **Detecting Sharing Patterns:** The sharing pattern between threads is tracked by using the data sampling features of the hardware PMU. For each thread, a summary vector, called *shMap*, is created in software that provides a signature of data regions accessed by the thread that resulted in cross-chip communication.

3. **Thread Clustering:** Once sufficient data samples are collected, the shMaps are analyzed. If threads have a high degree of data sharing then they will have similar shMaps and as a result, they will be scheduled to execute on the same cluster.

4. **Thread Migration:** The operating system scheduler attempts to migrate threads so that threads of the same cluster are as close together as possible.

We apply these phases in an iterative process. That is, after the thread migration phase, the system returns to the stall breakdown phase to monitor the effect of remote cache accesses on system performance and may re-cluster threads if there is still a substantial number of remote accesses. Application phase changes are automatically accounted for by this iterative process.

In the following subsections, we present the details of each phase.

3.4.1 Monitoring Stall Breakdown

Before starting to analyze thread sharing patterns, we determine whether there is a high degree of cross-chip communication with significant impact on application performance by looking at the stall breakdown. Thread clustering will be activated only if the share of remote cache accesses in the stall breakdown is higher than a certain threshold. Otherwise, the system continues to monitor the stall breakdown. We used an activation threshold of 20% and we evaluate it every billion processor cycles. That is, for every one billion processor cycles, if 20% of the cycles are stalled due to accessing remote caches, then the sharing detection phase is entered. Note that the overhead of monitoring stall breakdown is negligible since it is mostly done by the hardware PMU. As a result, we can afford to continuously monitor stall breakdown with no visible effect on system performance.

3.4.2 Detecting Sharing Patterns

In this phase, we monitor the addresses of the cache lines that require cache coherence actions with remote caches and construct a summary data structure for each thread, called *shMap*. Each shMap shows which data items each thread has fetched from caches on remote chips. We later compare the shMaps with each other to identify threads that are actively sharing data and cluster them accordingly.

Constructing shMaps

Each shMap is essentially a vector of 8-bit wide saturating counters. We believe that this size is adequate for our purposes because we are using sampling and are only looking for a rough approximation of sharing intensity. Each vector is given only 256 of these counters so as to limit overall space overhead. Each counter corresponds to a *region* in the virtual address space. Larger region sizes result in larger application address space coverage by the shMaps, but less precision and an increase in falsely reported sharing incidents. The largest region size with which no false-positives can occur is the size of an L2 cache line, which is the unit of data sharing for most cache-coherence protocols. Consequently, we used a region size of 128 bytes, which is the cache line size of our system.

With shMaps, we have effectively divided the application address space into regions of a fixed size. Since 256 entries at 128-byte region granularity is inadequate to cover an entire virtual address space, we made use of hashing. We used a simple hash function to map these regions to corresponding entries in the shMap. A shMap entry is incremented only when the corresponding thread incurs a remote cache access on the region. Note that threads that share data but happen to be located on the same chip will not cause their shMaps to be updated as they do not incur any remote cache accesses.

We rely on PMU hardware support to provide us with the addresses of remote cache accesses. While this feature is not directly available in most hardware PMUs, we use an indirect method to

capture the address of remote cache accesses with reasonable accuracy. In Section 3.5.1 we provide details of how we implemented this method on the IBM POWER5 processor.

Constructing shMaps involves two challenges. First, to record and process every remote cache access has high overhead, and second, with a small shMap, the potential rate of hash collisions may become very high. We use sampling to deal with both challenges. To cope with the overhead, we use *temporal sampling*. To deal with the high hash collision rate and eliminate the resulting aliasing problems, we use *spatial sampling*. Using temporal and spatial sampling of remote cache accesses instead of capturing them precisely is sufficient for our purposes because we only need an indication of the thread sharing pattern. If a data item is highly shared (i.e., remote cache accesses occur very frequently), it will likely be captured by the sampling. The overhead of this phase, for various temporal sampling rates, is measured in Section 3.6.3.

Temporal Sampling We record and process only one in N occurrences of remote cache access events. In order to avoid undesired repeated patterns, we constantly re-adjust N by a small random value. Moreover, the value of N is further adjusted by taking two factors into account: (i) the frequency of remote cache accesses, which is measured by the PMU, and (ii) the runtime overhead. A high rate of remote cache accesses allow us to increase N, since we will obtain a representative sample of addresses even with large values of N.

Spatial Sampling Rather than monitor the entire virtual address space, we select a small set of regions to be monitored for remote cache accesses. The regions are selected somewhat randomly, but there must be at least one remote cache access on a region to make it eligible to be selected. The hypothesis is that once a high level of sharing is detected on a subset of cache lines, it is a clear indication that the actual intensity of sharing is high enough to justify clustering.

We implement spatial sampling by using a filter to select remote cache access addresses after applying the hashing function. This *shMap filter*, as shown in Figure 3.4, is essentially a vector of addresses with the same number of entries as a shMap. All threads of a process use the same shMap filter. A sampled remote cache access address is considered further (i.e., is allowed to pass the filter) only if its corresponding entry in the shMap filter has the same address value. Otherwise, the remote cache access is discarded and not used in the analysis. Each shMap filter entry is initialized, in an immutable fashion, by the first remote cache access that is mapped to the entry. That is, threads compete for entries in the shMap filter and the first thread to access an entry determines the entry's value. This policy eliminates the problem of aliasing due to hash collisions. Figure 3.4 shows the function of the shMap filter. The numbered circles represent remote cache accesses, ordered by their indicated number. The first two accesses (#1 and #2) are able to pass through the filter and increment the appropriate vector entries. In contrast, the next two accesses (#3 and #4) are unable to pass the filter and cannot increment their vector entries because their addresses do not match the already configured shMap filter entries.

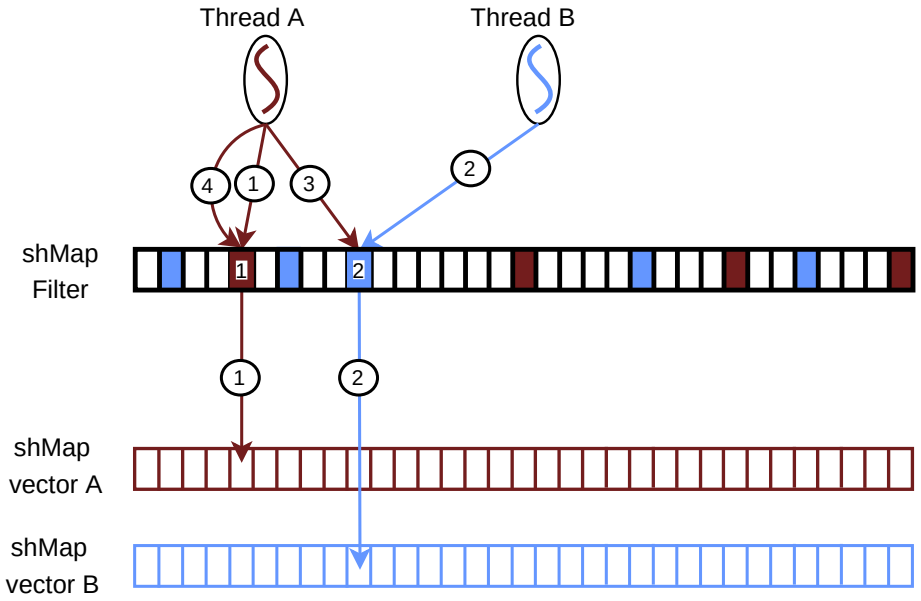


Figure 3.4: Constructing shMap vectors. To implement spatial sampling, each remote cache access by a thread is indexed into the shMap filter via a hash function. To eliminate aliasing due to hash collisions, only those remote cache accesses that pass the filter are allowed to increment the corresponding entry in the vector. The numbered circles represent remote cache accesses, ordered by their indicated number.

In an unlikely pathological case, it is possible that some threads starve out others by grabbing the majority of the shMap filter entries, thus preventing remote cache accesses of other threads from being recorded. We place a limit on the number of entries allowed by a thread to partially address this problem. For instance, in an application that has N threads, each thread is allowed to initialize no more than $\frac{1}{N}$ of the shMap filter entries. Additionally, we envision the thread clustering process to be iterative, thereby automatically handling insufficient thread clustering and shMap filter biasing in subsequent iterations. That is, after detecting sharing among some threads and clustering them, if there is still a high rate of remote cache accesses, the shMap vectors and filter are cleared of previous values, the entire thread clustering mechanism is activated again, and the previously victimized threads will obtain another chance. This clearing of the shMap vectors and filter can prevent the vector entries from being blocked out by useless data at the expense of losing more valuable data, such as during the startup phase of an application when data access and sharing patterns may substantially differ from steady-state application execution.

3.4.3 Thread Clustering

The forming of thread clusters requires a metric to measure similarity among threads. We first describe this metric, followed by our simple, one-pass algorithm for grouping threads.

Similarity Metric

We define the similarity of two shMap vectors, corresponding to two threads, as their dot products:

$$similarity(T_1, T_2) = \sum_{i=0}^{N-1} T_1[i] \times T_2[i] \tag{3.1}$$

where T_x is the shMap vector of thread x, and i is the i^{th} entry of the vector consisting of N entries. The rationale behind choosing this metric for similarity is two-fold. First, it automatically takes into account only those entries where both vectors have non-zero values. Note that T_1 and T_2 have non-zero values in the same location only if they have had remote cache accesses on the same cache line (i.e., the cache line is being shared actively). We consider very small values (e.g., less than 3) to be zero as they may be incidental or due to cold sharing and may not reflect a real sharing pattern.

Second, the metric takes into account the intensity of sharing by multiplying together the number of remote cache accesses each of the participating threads incurred on the target cache line. That is, if two vectors have a large number of remote cache accesses on a small number of cache lines, the similarity value will be large, correctly identifying that the two threads are actively sharing data. Equivalently, it can detect if two vectors have a low number of remote cache accesses but on a large number of cache lines, or any other combination within the two ends of the spectrum. Other similarity metrics could be used, but we found this metric to work quite well for the purpose of thread clustering.

In our experiments, we used a similarity threshold value of approximately 40,000. For two candidate vectors, this similarity threshold could be achieved under various simple scenarios, such as: (1) a single corresponding entry in each vector has values greater than 200; or (2) two corresponding entries in each vector have values greater than 145. In the first scenario, a single memory region is intensely shared between the two threads since each thread reports a sharing intensity value of 200 out of 255 for the memory region. In the second scenario, there are two memory regions that are strongly shared between the two threads since each thread reports a sharing intensity of 145 out of 255 for two memory regions, which is 57% of the maximum intensity value.

Forming Clusters

One way to cluster threads based on shMap vectors is to use standard machine learning algorithms, such as hierarchical clustering or k-means [Jain et al. 1999]. Unfortunately, such algorithms are too computationally expensive to be used online in systems with potentially hundreds or thousands of active threads, or they require the number of clusters to be known in advance, which is not realistic in our environment.

To avoid high overhead, we use a simple heuristic for clustering threads based on two assumptions that are simplifying but fairly realistic. First, we assume that data is naturally partitioned according to the application logic, and threads that work on two separate partitions do not share much except for data that is globally (i.e., process-wide) shared among all threads. In order to remove the effects of globally shared data on clustering, we ignore information on globally shared

cache lines when composing clusters. To determine which entries in the shMap vectors correspond to globally shared cache lines, we construct a histogram vector based on data from all shMap vectors. Each entry in this histogram vector indicates how many of the shMap vectors have a non-zero value in the corresponding entry. We then use a threshold value of N to classify each entry in the histogram vector as either globally shared ($\geq N$) or locally shared ($< N$). In our experimental results, we used a threshold value of $N = \frac{1}{2} \times number\_of\_shMap\_vectors$. That is, we consider a cache line to be globally shared if more than half of the total number of threads have incurred a remote cache access on it. In contrast, we found that requiring all threads to incur a remote access to the corresponding cache line to be too stringent, as our relatively narrow monitoring window might only capture a fraction of threads performing such sharing. For example, the observed fraction may be 90%, 80%, 75%, but we found that using a threshold value corresponding to 50% to work quite well. A value of 50% clearly identified globally shared cache lines without being overly sensitive to the precise fraction observed within the monitoring window.

The second assumption in our simple heuristic for clustering threads is that when a subset of threads share data, the sharing is reasonably symmetric. That is, it is likely that *all* threads of the subset incur remote cache accesses on similar cache lines, no matter how they are partitioned.

As a result, the clustering algorithm can be simplified as follows. Based upon the first assumption, if the similarity between shMap vectors is greater than a certain threshold, we consider them to belong to the same cluster. According to the second assumption, any shMap vector can be considered as a cluster representative since all elements of a cluster share common data equally strongly.

The clustering algorithm scans through all threads in one pass and compares the similarity of each thread with the representatives of known clusters. If a thread is not similar to any of the known cluster representatives, a new cluster is created, and the thread that is currently being examined is designated as the representative of the newly created cluster. The set of known clusters is empty at the beginning.

The computational complexity of this algorithm is $O(T * c)$ where T is the number of threads that are suffering from remote cache accesses, and c is the total number of clusters, which is usually much smaller than T.

3.4.4 Thread Migration

Once thread clusters are formed, each cluster is assigned to a chip with the global goal of maintaining load balance. That is, in the end, there should be an equal number of threads on each chip. Our cluster-to-chip assignment strategy consists of the following. Naturally, the initial condition is that all chips start with no threads assigned to them[2]. First, we sort the clusters from the largest size to the smallest size so that we can easily select the next largest available cluster to migrate. Second,

[2]The only exception is for threads that have been labelled by the operating system as "non-migratable".

we use a simple, one-pass, greedy assignment algorithm that traverses this sorted list from largest to smallest cluster and assigns each cluster to the chip with the currently lowest number of assigned threads.

To handle a scenario that would lead to thread imbalance, the following condition is checked at each loop iteration of the algorithm. At each iteration, based upon a quick and simple calculation, if the proposed assignment of the candidate cluster would, at the end of the algorithm, cause an imbalance among chips, we instead neutralize this cluster by evenly distributing its threads among all chips. For example, consider a system containing 2 chips and currently no threads assigned to either chip. If the currently considered cluster contains 100 threads and the remaining clusters (and non-clusters) of threads contain a total of 10 threads, it would not be possible to achieve a balanced system at the end if we assign the 100 thread cluster to one chip and assign the remaining 10 threads to the other the chip. To perform this check on our 2 chip system, we simply compare the number of threads on each chip, assuming that the currently considered cluster is assigned to the chip with the least number of threads and that any remaining unassigned threads are assigned to the other chip. If the resulting number of threads on each chip varies greatly, such as by more than 25%, then the currently considered cluster is neutralized by evenly distributing its threads among all chips.

Once all thread clusters have been considered, the remaining non-clustered threads are assigned to the chips to balance out any remaining differences[3]. We recognize that this algorithm is a best-effort, practical, online strategy that provides no guarantee of optimality.

Load balance within each chip is addressed by uniformly and randomly assigning threads to the cores and the different hardware contexts on the core. To minimize cache capacity and conflict problems within a single chip, a variety of intra-chip scheduling techniques described in Section 3.2 could be applied, such as the schedulers proposed by Fedorova et al. [Fedorova et al. 2005 2006], and Bulpin and Pratt [Bulpin and Pratt 2005], although we have not applied these techniques in this dissertation.

In balancing threads among chips, cores, and hardware contexts, we make the simplifying assumption that threads are fairly homogeneous in their usage of assigned scheduling quantum. Although we have not done so in our work, default Linux load balancing within each chip could be enabled, as opposed to load balancing across chips, so that balancing can take place among the cores and hardware contexts within a chip. This feature could help in reducing the severity of any subsequent load imbalance within the chip.

[3] "Non-migratable" threads are not migrated.

Item	Specification
# of Chips	2
# of Cores	2 per chip
CPU Cores	IBM POWER5, 1.5 GHz, 2-way SMT
L1 ICache	64 kB, 2-way set-associative, per core
L1 DCache	32 kB, 4-way set-associative, per core
L2 Cache	1.875 MB, 10-way set-associative, per chip
L3 Victim Cache	36 MB, 12-way set-associative, per chip, off-chip
RAM	8 GB (2 banks × 4 GB)

Table 3.1: IBM OpenPower 720 specifications.

3.5 Experimental Setup

The multiprocessor used in our experiments is an IBM OpenPower 720 computer. It is an *8-way* POWER5 machine consisting of a 2×2×2 SMP×CMP×SMT configuration, as shown in Figure 3.1[4]. Table 3.1 describes the hardware specifications.

We used Linux 2.6.15 as the operating system. Linux was modified in order to add the features needed for hardware performance monitoring, including the stall breakdown and remote cache access address sampling. We also changed the scheduler code to migrate threads according to our thread clustering scheme. Our modifications consist of approximately 200 lines of code (LOC), stemming from 50 LOC for detecting sharing, 60 LOC for clustering threads, and 90 LOC for migrating threads.

3.5.1 Platform-Specific Implementation: Capturing Remote Cache Accesses on POWER5

The POWER5 PMU cannot directly capture remote cache accesses with a single PMU event (and register), but rather, requires two PMU events (and registers) to obtain this information. For the first PMU event, we configure the POWER5 PMU to make use of its *continuous data sampling* mechanism in order to capture the data address of every local L1 data cache miss in a continuous fashion regardless of the instruction that caused the data cache miss. The data address is recorded in a PMU register which is updated upon every data cache miss. Unfortunately, it is not possible to directly determine whether the sampled local L1 data cache miss was satisfied by a remote or local cache access.

In order to filter these cache accesses and obtain only the remote cache accesses, a second PMU event (and register) is used in parallel. This second PMU event (and register) *counts* the occurrences of local L1 data cache misses that are resolved by a remote L2 or remote L3 cache access. We configure this second PMU register to raise an overflow exception, and thus freeze all PMU registers, when a certain number of remote cache accesses has been reached. Once an

[4] 2 chips × 2 cores per chip × 2 hardware threads per core.

overflow exception is raised, the "last" local L1 data cache miss captured by the first PMU event (and register) is highly likely to have required a remote cache access that caused the second PMU register to overflow.

Therefore, by reading the sample data register (from the first PMU register) only when the remote cache access counter overflows (from the second PMU register), we ensure that most of the samples read are actually remote cache accesses. Requiring two parallel, uncoupled events to occur in a specific order and within a specific time-frame creates a situation that contains data races between the two PMU events and also between these two targetted events and other microarchitectural events. Fortunately, our experiments with various microbenchmarks verify the effectiveness of this method as almost all of the local L1 data cache misses recorded in our trace are indeed satisfied by remote cache accesses.

3.5.2 Workloads

For our experiments, we used a synthetic microbenchmark and three commercial server workloads: VolanoMark 2.5.0.9 [Volano], which is a Java-based Internet chat server workload, SPECjbb2000 [SPEC], which is a Java-based application server workload, and RUBiS [RUBiS], which is an online transaction processing (OLTP) database workload. For VolanoMark and SPECjbb, we used the IBM J2SE 5.0 Java virtual machine (JVM). For RUBiS, we used MySQL 5.0.22 as our database server. These server applications are written in a multithreaded, client-server programming style, where there is a thread to handle each client connection for the life-time of the connection. We present details of each benchmark below.

Synthetic Microbenchmark

The synthetic microbenchmark is a simple multithreaded program in which each worker thread reads and modifies a scoreboard. Each scoreboard is shared by several threads, and there are several scoreboards. Each thread has a private region of data to work on which is fairly large so that accessing it often causes data cache misses. We use this configuration to verify that our technique is able to distinguish remote cache accesses that are caused by accessing the shared scoreboards from local cache accesses that are caused by accessing the private data. All scoreboards are accessed by a fixed number of threads. A clustering algorithm is supposed to cluster threads that share a scoreboard and consider them as the unit for thread migration.

VolanoMark

VolanoMark is an instant messaging chat server workload. It consists of a Java-based chat server and a Java-based client driver. The number of rooms, number of connections per room, and client think times are configurable parameters. This server is written using the traditional, multithreaded, client-server programming model, where each connection is handled completely by a designated pair

of threads for the life-time of the connection. Given the nature of the computational task, threads belonging to the same room should experience more intense data sharing than threads belonging to different rooms.

In our experiments, we used 2 rooms with 8 clients per room and 0 think time as our test case. In this setting, the hand-optimized placement of threads would be for the threads belonging to the same room to be located on the same chip. From another perspective, each room should be assigned to a separate chip. In the worst case scenario, the threads are placed randomly or in a round-robin fashion.

SPECjbb2000

SPECjbb2000 is a self-contained Java-based benchmark that consists of multiple threads accessing designated *warehouses*. Each warehouse is approximately 25 MB in size and stored internally as a B-tree variant. Each thread accesses a specified warehouse for the life-time of the experiment. Given the nature of the computational task, threads belonging to the same warehouse should experience more intense data sharing than threads belonging to different warehouses.

In our experiments, we modified the default configuration of SPECjbb so that multiple threads can access a warehouse. Thus, in our configuration, we ran the experiments using 2 warehouses and 8 threads per warehouse.

RUBiS

RUBiS is an OLTP database server workload that represents an online auction site workload in a multi-tiered environment. The client driver is a Java-based web client that accesses an online auction web server. The front-end web server uses PHP to connect to a back-end database. In our experiments, we ran MySQL 5.0.22 as our back-end database. We focus on the performance of the database server. We made a minor modification to the PHP client module so that it uses persistent connections to the database, allowing for multiple MySQL requests to be made within a connection. While this modification improves performance by reducing the rate of TCP/IP connection creation and corresponding thread creation on the database server, it also enables our algorithm to monitor the sharing pattern of individual threads over the long term.

In our workload configuration, we used two separate *database instances* within a single MySQL process. We used 16 clients per database instance with no client think time. This configuration may represent, for instance, two separate auction sites run by a single large media company. We expect that threads belonging to the same database instance will experience more intense sharing with each other than with other threads in the MySQL process. We ran two client driver instances, each making requests to its corresponding auction site. It should be noted that the persistent connections are reused within each auction site and not across auction sites. This guarantees that the database thread that handles the connection will only see requests for only one of the *databases*

rather than a mix of both *databases*.

3.5.3 Thread Placement

We evaluated four thread placement strategies: default Linux, round-robin, hand-optimized, and automatic thread clustering. The default Linux thread placement strategy attempts to find the least loaded processor in which to place the thread. In addition, Linux performs two types of dynamic load balancing: *reactive* and *pro-active*. In reactive load balancing, once a processor becomes idle, a thread from a remote processor is found and migrated to the idle processor. Pro-active load balancing attempts to balance the processor time each threads receives by automatically balancing the length of the processor run queues. The default Linux scheduler does not take data sharing into account when migrating and scheduling the threads.

For round-robin scheduling, we modified Linux to disable dynamic load balancing. Threads of our targeted workload are placed in a round-robin fashion among processors. This thread placement strategy is unaware of data sharing patterns among threads. The round-robin scheduling is implemented in order to be able to exhibit worst case scenarios where sharing threads are scattered onto different chips.

With hand-optimized scheduling, threads are placed by considering natural data partitioning according to the application logic[5]. For VolanoMark, threads belonging to one room are placed onto one chip while threads belonging to the other room are placed onto the other chip. Within each chip, threads of the room are placed in a round-robin fashion to achieve load balance within the chip. Similarly for SPECjbb, threads of one warehouse are placed onto the same chip. The same pattern applies for RUBiS: the threads of one database instance are placed onto one chip while threads of the second database instance are placed onto the other chip. For hand-optimized scheduling, the Linux scheduler is modified to disable both reactive and pro-active load balancing.

In the next section, we evaluate and analyze how closely our automated scheme comes to achieving the performance of hand-optimized placement.

3.6 Results

We first visualize the effectiveness of thread clustering in detecting sharing. We then show the impact on performance of clustering the sharing threads. We also briefly examine the runtime overhead of temporal sampling and the impact on clustering due to spatial sampling.

3.6.1 Thread Clustering

Figure 3.5 shows a visual representation of shMap vectors and the way they have been clustered for the four applications. The shMap vectors, which were depicted in detail in Figure 3.4, have been

[5]We do not claim that the hand-optimized thread placements are the optimal placements, but are merely significantly improved placements based on application domain knowledge.

grouped in Figure 3.5 by cluster, along the y-axis according to our thread clustering algorithm. Since shMap vectors are laid out horizontally in the graphs, the reader may simply think of the vectors effectively as the memory regions of each thread laid out horizontally. The x-axis very roughly represents the virtual address space divided into memory regions[6]. For each thread (shMap vector), each grey-scale dot indicates the intensity of cross-chip sharing of the corresponding memory region with other threads. The darker a dot is, the more often remote cache accesses have been sampled for the corresponding region.

For each cluster, we can see several memory regions that are shared by all threads within the cluster but not with threads of other clusters. These shared regions appear as dark vertical line segments that span only its cluster. As an example, in Figure 3.5b, where SPECjbb was configured with 4 warehouses and 16 threads per warehouse, we see 4 clusters, each consisting of 16 shMap vectors that correspond to 16 threads accessing a particular warehouse[7]. Each cluster contains several dark vertical line segments that span only that particular cluster, indicating shared memory regions containing the per-warehouse data that is exclusively accessed by its threads.

To further simplify the graphs, the globally (process-wide) shared data have been removed, which would have appeared as dark vertical line segments spanning the entire length of the y-axis. Although Figure 3.5 is shown *after* thread clustering has been applied, to visualize the appearance of the graphs before thread clustering is applied, imagine a random, ungrouped ordering of the shMap vectors (threads) that causes each graph to resemble "snow" on a television screen.

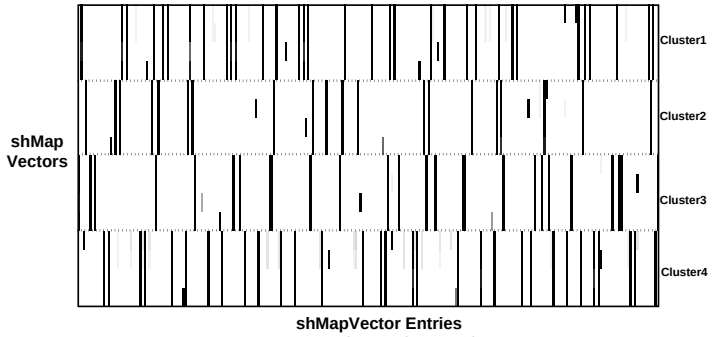
From Figure 3.5 it is clear that the shMaps are effective in detecting sharing and for clustering threads for three applications out of four (microbenchmark, SPECjbb, and RUBiS). In all three cases the automatically detected clusters conform to a manual clustering that can be done with specific knowledge about the application logic (i.e., a cluster for each scoreboard for the microbenchmark, for each warehouse in SPECjbb, and for each database instance in MySQL). JVM garbage collector threads in SPECjbb and VolanoMark did not affect cluster formation since they are run infrequently and do not have the opportunity to exhibit much sharing.

For VolanoMark however, the detected clusters do not conform with the logical data partitioning of the application logic (i.e., one partition per chat room). However, as we will show later, the automatic clustering approach still improves performance by co-locating threads that share data.

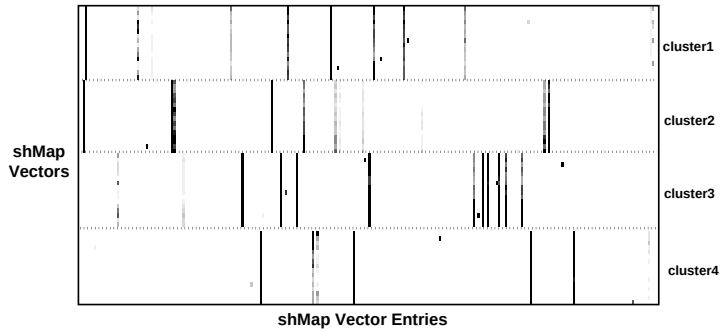
Given the nature of the sharing patterns of our workloads, a potential anomaly appears in the graphs. For example, in Figure 3.5b for SPECjbb, in addition to the dark vertical line segments that span each cluster of threads, there are some darkly-shaded dots that do not form a vertical line segment spanning the cluster. Each dot indicates a shared memory region of a particular thread that requires remote cache accesses, however, it appears that other threads of its cluster do not suffer from the same remote cache access requirements. One would expect some form of

[6]In actuality, the x-axis represents a hashing of the virtual address space, as described in Section 3.4.2.

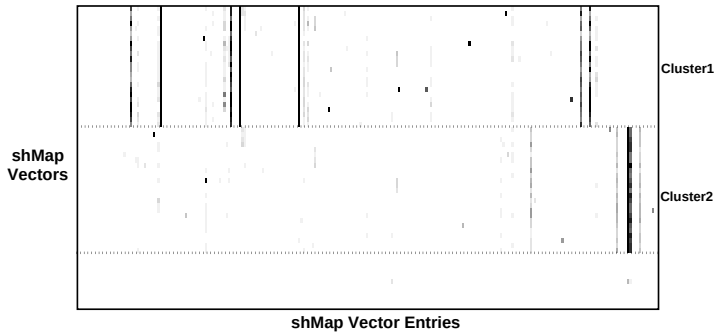
[7]For illustration purposes, SPECjbb was run with 4 warehouses. In subsequent experiments, 2 warehouses are used.



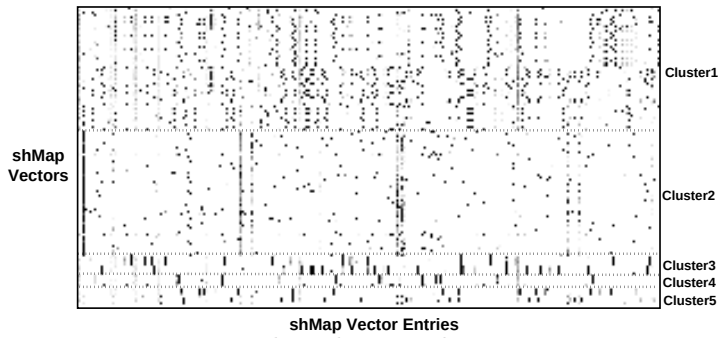
a. Microbenchmark



b. SPECjbb2000



c. RUBiS



d. VolanoMark

Figure 3.5: A visual representation of shMap vectors grouped by cluster along the y-axis according to our thread clustering algorithm. Each labelled cluster consists of several *rows* of shMap vectors. Since shMap vectors are laid out *horizontally*, the reader may simply think of them effectively as the memory regions of each thread laid out horizontally. The x-axis (shMap vector entries) very roughly represents the virtual address space divided into memory regions. For each thread (shMap vector), each grey-scale dot indicates the intensity of cross-chip sharing of the corresponding memory region with other threads. The darker a dot is, the more often remote cache accesses have been sampled for the corresponding region.

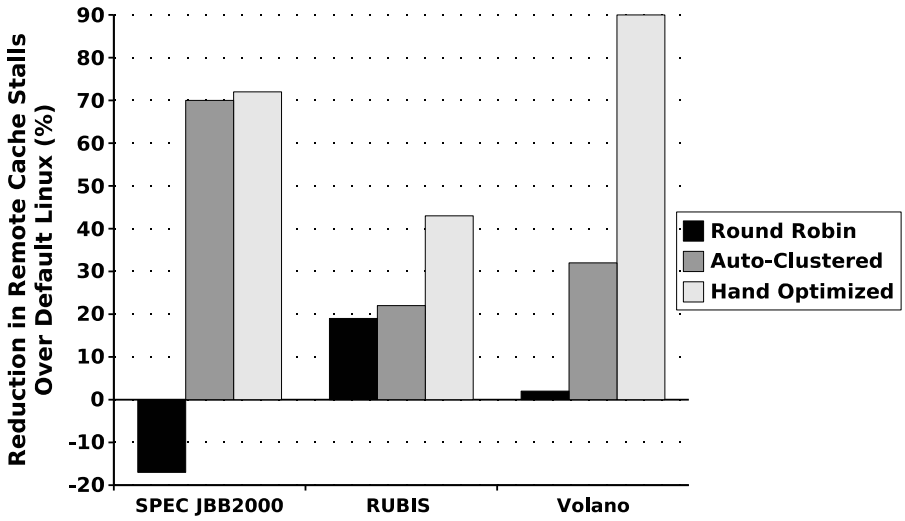


Figure 3.6: The impact of the scheduling schemes on reducing processor pipeline stalls caused by remote cache accesses. The baseline is Linux default scheduling. Higher y-axis values are better. It is possible to remove a significant portion of remote access stalls either by hand-optimizing the thread placement or through automatic clustering.

symmetry to exist in the graphs, such as having at least one other thread of the cluster showing a similar darkly shaded dot, because threads sharing a region should experience a similar number of remote cache accesses. However, there is the possibility that while one thread accesses the targetted shared region via a remote cache, the other thread accesses the targetted shared region via main memory. This main memory access, rather than remote cache access, occurs when the targetted shared region has already been evicted from the remote cache due to a variety of reasons, such as insufficient capacity in a particular cache-set of the set-associative cache.

It is interesting to note that the visualizations created in Figure 3.5, before or after clustering, could potentially be used as a stand-alone performance debugging tool for multithreaded workloads. For example, this performance monitoring tool could be used to examine the impact of a manually-specified, programmer-directed distribution of threads among multiple chips. This tool could also be used to verify the expected communication pattern of these threads.

3.6.2 Performance Results

Figure 3.6 shows the impact of the different thread scheduling schemes on processor pipeline stalls caused by accessing high-latency remote caches. In general, it is clear that it is possible to remove a significant portion of remote access stalls either by hand-optimizing the thread placement, or through automatic clustering. For SPECjbb, the automatic clustering approach performs nearly as well as the hand-optimized method. For the other two applications there is still further room for improvement.

Figure 3.7 shows the impact of the different thread scheduling schemes on application performance – for SPECjbb, the application-reported throughput in terms of operations/second; for

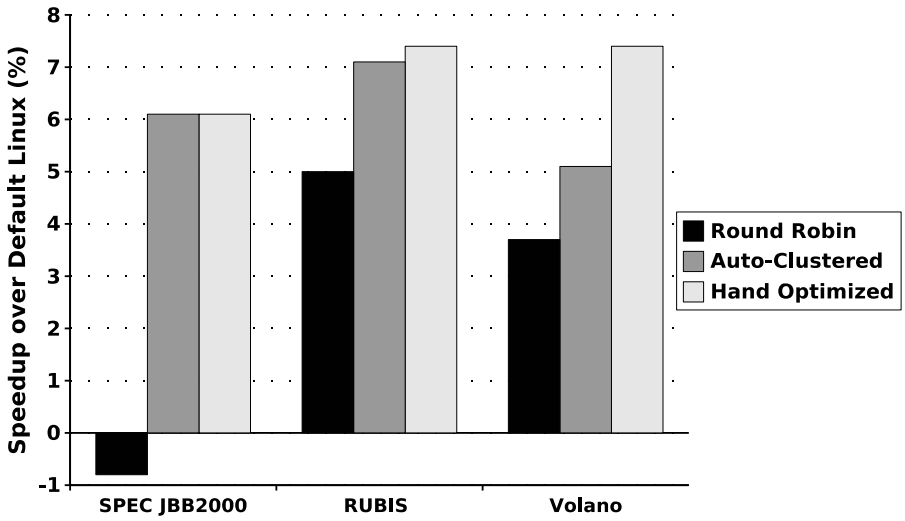


Figure 3.7: The impact of scheduling schemes on application performance. The baseline is Linux default scheduling. Both the hand-optimized and the automatic clustering schemes manage to improve performance by a reasonable amount.

VolanoMark, the application-reported throughput in terms of messages/second; and for RUBiS, the IPC (instructions-per-cycle) reported by the hardware PMU. Again, both the hand-optimized and the automatic clustering schemes manage to improve performance by a reasonable amount, but there is still room for improving the automatic clustering scheme. The magnitudes of performance gain appear reasonable because they approximately match the reduction in processor stalls due to remote cache accesses. For example, in Figure 3.3, 6% of stalls in VolanoMark were due to remote cache accesses and thread clustering was able to improve performance by 5% by removing most of these stalls.

For some workloads, these performance gaps may be due to the fact that the automatically chosen clusters do not precisely match the manually chosen clusters. For example, the automatically chosen clusters of the RUBiS workload, shown in Figure 3.5c, did not result in precisely 2 clusters but 2 clusters and several remaining, unclusterizable threads. For VolanoMark, Figure 3.5d indicates that 5 clusters were detected rather than 2. These unclustered and erroneously clustered threads may be responsible for the additional remote cache stalls indicated in Figure 3.6.

3.6.3 Runtime Overhead and Temporal Sampling Sensitivity

The average runtime overhead for identifying stall breakdown is negligible, around 2% for a sampling frequency of 20,000 samples per second [Azimi et al. 2005]. Therefore, the main runtime overhead of the system is due to detecting sharing patterns and thread migration. Figure 3.8 shows the runtime overhead of the sharing detection phase as a function of temporal sampling rate in terms of the percentage of the remote cache accesses that are actually examined for SPECjbb. As a higher percentage of remote cache accesses are captured, overhead increases. However, the length of this phase is fairly limited and only lasts until we have collected a sufficient number of samples

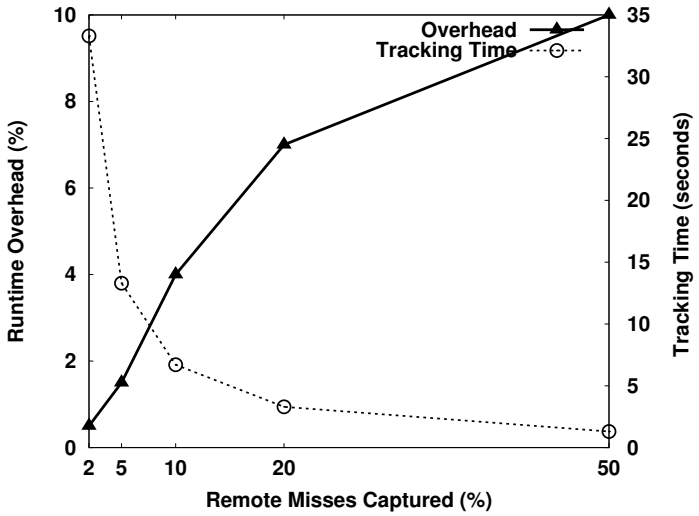


Figure 3.8: For SPECjbb, the runtime overhead of the sharing detection phase and the time that is required to collect a million remote cache access samples. The x-axis is the temporal sampling rate, in terms of the percentage of the remote cache accesses that are sampled. Lower y-axis values are better. A remote cache access sampling rate of 10% is a good point in the trade-off spectrum between overhead and collection time latency, resulting in a tracking overhead of 4% for a duration of 6.7 seconds.

to be able to cluster the threads. In our experiments, we have found we need roughly a million samples to accurately cluster the threads. Therefore, on the right y-axis of Figure 3.8, we show how long we need to stay in the detection phase to collect a million samples. Hence, the higher the sampling rate, the higher the run-time overhead will be, but the shorter the detection phase will last. According to Figure 3.8 it seems a sampling rate of 10% (capturing one in every 10 remote cache accesses) is a good balance point in this trade-off, resulting in a tracking overhead of 4% for a duration of 6.7 seconds.

With these overheads and sharing detection latencies in mind, automated thread clustering is worthwhile for multithreaded workloads that consist of long-lived threads (longer than X seconds, e.g., 6.7 seconds in previous our example), where threads exhibit phases of clustered data sharing patterns for at least X seconds in duration. This minimum time is required to recover the overhead costs and "break-even", with any addition time used to reap the benefits of the improved performance. Examples of such multithreaded workloads may include: database data mining and decision support system (DSS) workloads, similar in behaviour to the TPC-H benchmark workload; suitably designed game server workloads; and long-running scientific workloads that exhibit clustered data sharing patterns among its threads. Automated thread clustering is not suitable for multithreaded workloads where threads are short-lived, as this scheme would only add net overhead. In a realistic computing environment, the operating system could allow the user or system administrator to enable or disable automated thread clustering of the application.

3.6.4 Spatial Sampling Sensitivity

Although not shown, we have tried varying the number of entries in the shMap vectors for our workloads and found the cluster identification to be largely invariant. For example, we ran experiments using shMap sizes of 128 entries and 512 entries. The impact of using 128 entries as opposed to 256 entries on SPECjbb can be roughly visualized by covering the left half of the Figure 3.5b. Clustering would still identify the same groups of threads as sharing.

3.7 Discussion

In this section, we consider the issues of local cache contention, migration costs, requirements of the PMU, and the properties of hardware.

3.7.1 Local Cache Contention

Clustering too many threads onto the same chip could create local cache contention problems. The local caches may not have sufficient capacity to contain the aggregate working set of the threads. In addition, because these local caches are not fully associative but are set-associative, cache conflict problems may be magnified. Thus, the operating system scheduler may need to take these factors into consideration. Fortunately in our system, local L2 cache contention is mitigated by a large local L3 cache (36 MB). Nevertheless, local cache contention was not significant in our workloads.

3.7.2 Migration Costs

Thread migration incurs the costs of cache context reloading into the local caches and TLB (translation look-aside buffer) reloading. Compared to typical process migration that is performed by operating systems, such as default Linux, thread migration has lower costs since threads in a single application address space typically exhibit more cache context and TLB sharing. Any reloading costs are expected to be amortized over the long thread execution time at the new location, where threads enjoy the benefits of reduced remote caches accesses. Our results in Section 3.6.2 have shown scenarios where these benefits outweigh the costs.

3.7.3 PMU Requirements

Ideally, we would like the ability to specifically configure the PMU to continuously record the data address of remote cache accesses. Unfortunately, this direct capability is not available on the POWER5 processor and so it was composed using basic PMU capabilities as described in Section 3.5.1.

The Intel Itanium 2 PMU has low-level features that could be used in combination to allow for thread clustering, based upon the description given by Buck and Hollingsworth, Lu et al., and Marathe and Mueller [Buck and Hollingsworth 2004; Lu et al. 2004; Marathe and Mueller

2006]. On Intel x86 processors, it may be possible to use the Intel PEBS (Precise Event-Based Sampling) PMU to indirectly capture the required information [Sprunt 2002]. Instead of directly capturing the data address of the remote cache access, it may be possible to use PEBS to capture the entire architectural state of the processor on each remote cache access. With this information, the instruction pointer must be followed to its instruction, which must be then disassembled to determine which architectural register contains the data address of the remote cache access. On other IBM POWER/PowerPC and AMD processors which have instruction-based sampling, it may also be possible to obtain a sample of data addresses of remote cache accesses [Drongowski 2007; IBM 2005a]. With instruction-based sampling, selected types of instructions can be tagged and monitored as they traverse the processor pipeline. For example, load and store instructions could be tagged and monitored for remote cache accesses. For each such event, it may be possible to record the required information, such as the value of the architectural register containing the data address of the load.

It is interesting to note that although hardware designers initially added PMU functionality primarily to collect information for their own purposes, namely for designing the next generation of processor architectures, PMUs have become surprisingly useful for purposes other than those for which they were envisioned. Consequently, hardware designers are now adding more and more PMU capabilities requested by software designers. We hope that our work provides compelling evidence of the useful application of PMU sharing detection capabilities so that more processor manufacturers would seriously consider directly adding them to future processors.

3.7.4 Important Hardware Properties

Our thread clustering approach is viable because there exists a large disparity between local and remote cache latencies. On larger multiprocessor systems, where this disparity is even greater, we expect higher performance gains. In actuality, running on a *32-way* POWER5+ multiprocessor consisting of 8 chips, we saw a greater performance impact from thread clustering. Our results indicate a 14% throughput improvement in SPECjbb when comparing handcrafted placement to the default Linux configuration. Automatic thread clustering was not implemented on the 32-way POWER5+ system due to time constraints combined with a number of complexities, such as ownership of the 32-way POWER5+ belonging to IBM Toronto Labs, differences between the POWER5 PMU versus the POWER5+ PMU, and differences in the Linux kernel version used on the two systems. However, we believe that automatic thread clustering should achieve performance improvements similar to handcrafted placement.

3.8 Concluding Remarks

Following our principle of promoting the shared use of the last-level cache, we have described the design and implementation of a scheme to schedule threads based on sharing patterns detected

online using features of standard performance monitoring units (PMUs) available in modern processors. Experimental results indicate that our scheme is reasonably effective: running commercial multithreaded server Linux workloads on an *8-way* POWER5 SMP-CMP-SMT multiprocessor, our scheme was able to reduce processor pipeline stalls caused by cross-chip cache accesses by up to 70%, resulting in performance improvements of up to 7%. On a larger-scale multi-chip platform, consisting of a *32-way* IBM POWER5+ system, we found a maximum potential for up to 14% performance improvement.

We use operating system scheduling to promote shared use of the last-level cache. We match the sharing that occurs in software with the available hardware sharing facilities.

This work, we believe, represents the first time hardware PMUs have been used to detect sharing patterns in a fairly successful fashion. More specifically, we have found that our method of identifying sharing patterns using shMap signatures to be surprisingly effective for the purpose of promoting sharing by the operating system, considering (i) their relatively small size of only 256 entries, and (ii) the liberal application of sampling along two dimensions (temporal and spatial).

Chapter 4

Providing Isolation in the Shared Cache

"Better to be alone than in bad company." – Thomas Fuller

In some workload environments, sharing among processes or threads is not prevalent. An example would be a multiprogrammed, single-threaded computing environment, which consists of multiple applications that each have one thread of execution. In this situation, disparate processes, which do not share any data or instructions, executing on the cores share the cache in an unrestricted manner and may interfere with each other. Specifically, this situation can lead to cache line interference between non-sharing processes, resulting in significant performance degradation. A process may unintentionally evict cache lines belonging to a non-related process that is currently executing elsewhere on the multicore processor rather than evict one of its own cache lines. This scenario of non-shared use of a shared hardware resource leads us to our second shared-cache management principle of providing isolation in the shared cache. By providing cache space isolation, we circumvent a major disadvantage of shared caches, namely cache space interference among applications.

We demonstrate the application and effectiveness of the principle of providing isolation in this chapter. We control which processes or threads have access to which sections of the shared cache. We accomplish this task at the operating system level by controlling the allocation of physical pages. In effect, with a page coloring technique, we can flexibly partition a large shared cache into smaller private caches to provide space isolation capabilities. We demonstrate performance improvements of up to 17% in terms of instructions-per-cycle (IPC).

4.1 Introduction

Shared caches have important advantages such as increased cache space utilization, fast inter-core communication (via the high-speed shared L2 cache), and reduced aggregate cache footprint through the elimination of undesired replication of cache lines. However, a major disadvantage of

shared L2 caches is that uncontrolled contention can occur because the execution cores can freely access the entire L2 cache. As a result, scenarios can occur where one core constantly evicts useful L2 cache content belonging to another core without obtaining a significant improvement itself. Such contention causes increased L2 cache misses which in turn leads to decreased application performance.

Consider, for example, an MP3 player that streams through a lot of data without reuse. It severely and continuously pollutes the cache with an attendant drastic effect on the performance of the other applications running on the other cores of the chip. More generally, this example describes how I/O-bound applications can significantly interfere with the performance of cache-sensitive, processor-bound applications. It is worth noting again that the on-chip cache is a performance-critical hardware resource because a miss to this cache requires off-chip access, which typically costs an order of magnitude longer latency to complete. For example, on an IBM POWER5 processor, which has one of the lowest of such ratios, it requires 14 cycles to reach the on-chip L2 cache compared to 90 cycles to reach the off-chip L3 cache and 280 cycles to reach the off-chip main memory.

Uncontrolled L2 cache sharing also reduces the ability to enforce priorities and to provide quality-of-service (QoS). For example, a low-priority application running on one core that rapidly streams through the L2 cache can consume the entire L2 cache and remove most of the working set of higher-priority applications co-scheduled on another core. In terms of QoS, uncontrolled L2 cache sharing introduces performance instability or variability in multiprogrammed workloads, which is an undesired characteristic, where the nature of the variability depends on which other applications are running on other cores of the multicore processor and exactly which phases these other applications are in.

Many researchers in the architecture community have recognized the problem of uncontrolled contention in the L2 cache and have explored different hardware support for dynamically partitioning the L2 cache in order to provide cache space isolation [Chandra et al. 2005; Guo and Solihin 2006; Iyer 2004; Kannan et al. 2006; Kim et al. 2004; Qureshi and Patt 2006; Suh et al. 2004]. Some of these hardware solutions are effective and may eventually appear in future processors. We argue that an alternative, operating system-level solution is viable on existing multicore processors.

In this chapter, we present such a software solution to provide cache space isolation, based on a low-overhead and flexible implementation of cache partitioning through physical page allocation on a real operating system (Linux) running on a real multicore system (IBM POWER5).

In our experimental results, we show how our software cache partitioning mechanism can provide cache space isolation to eliminate the negative impact of uncontrolled sharing of the L2 cache. We used SPECcpu2000 and SPECjbb2000 as our workloads, running Linux 2.6.15 on an IBM POWER5 CMP system. Our experimental results indicate that by carefully partitioning the L2 cache and co-scheduling compatible applications appropriately, we can achieve performance improvements up of to 17%, in terms of IPC, stemming from the aggregate percentage improvement experienced by

each application in the multiprogrammed workload[1].

4.2 Related Work

Many researchers in the architecture community have recognized the cache contention problem in shared L2 caches and have proposed hardware support for partitioning the cache [Chandra et al. 2005; Guo and Solihin 2006; Iyer 2004; Kannan et al. 2006; Kim et al. 2004; Lin et al. 2009; Liu et al. 2004; Qureshi and Patt 2006; Rafique et al. 2006; Srikantaiah et al. 2008; Suh et al. 2004; Xie and Loh 2008 2010]. Others have proposed further hardware modifications to the cache-coherence protocol, or cache line eviction/insertion policies to achieve performance improvements in a shared cache [Jaleel et al. 2008; Liu et al. 2004; Liu and Yeung 2009; Srikantaiah et al. 2008; Zhang and Asanović 2005], or to attain the advantages of a shared cache but with private caches [Chang and Sohi 2006; Chishti et al. 2005; Yeh and Reinman 2005; Youn et al. 2007]. More radical, hierarchical re-organizations of the on-chip caches have also been proposed by several researchers in order to retain some of the benefits of shared caches while also adding benefits experienced by private caches [Beckmann and Wood 2004; Beckmann et al. 2006; Huh et al. 2005]. Some of these hardware solutions are effective, with reasonable complexity and resource consumption, and may eventually be implemented in real processors in the future.

Our work explores an alternative solution that is entirely based on software. Our software-based approach has the advantage of being implementable and deployable today. Moreover, it is more flexible and does not add to the design complexity of already complex microprocessors. While the hardware solution proposed by Qureshi and Patt can achieve up to 23% performance improvement on a simulated platform, our software-based solution running on a real system is able to achieve up to 17% improvement [Qureshi and Patt 2006].

The work closest to our approach is by Cho and Jin, who proposed a software-based mechanism for L2 cache partitioning based on physical page allocation [Cho and Jin 2006 2007]. However, the major focus of their work was on how to distribute data in a tile-based Non-Uniform Cache Architecture (NUCA) multicore chip to minimize overall data access latencies. In contrast, we concentrate solely on the problem of uncontrolled contention on a shared L2 cache, providing it with isolation properties. Furthermore, we have implemented our solution in a real environment based on features available on existing processors. This enables us to examine, using hardware PMUs, the impact of the cache partitioning on real processor performance. Similar to their philosophy, we advocate low-overhead, flexible software solutions that help to simplify the hardware. Due to

[1]Aggregate percentage improvement in IPC, where $IPC(i)$ is the average IPC of the i^{th} application of a multi-programmed workload consisting of N applications:

$$\sum_{i=0}^{N-1} \frac{IPC(i)_{new} - IPC(i)_{old}}{IPC(i)_{old}}$$

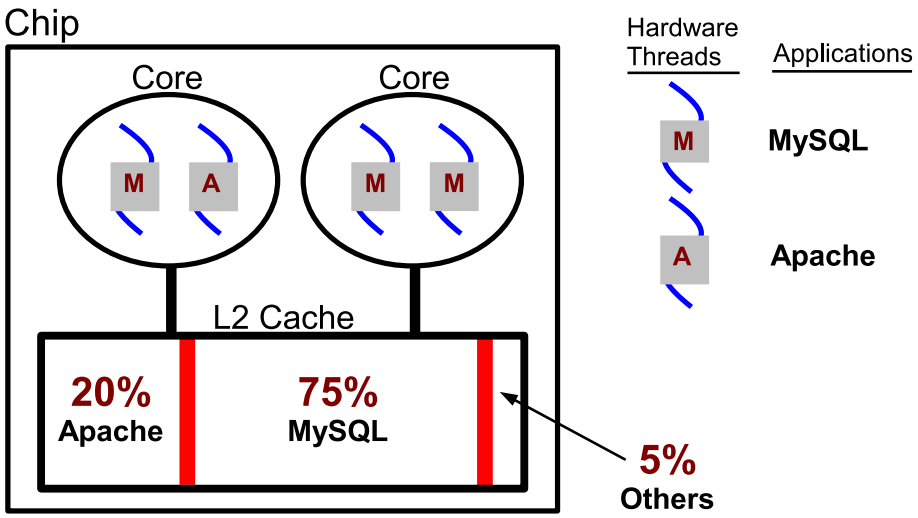


Figure 4.1: Providing isolation in a shared cache by partitioning it. 75% of the L2 cache space cache is allocated exclusively to the MySQL database application, 20% to the Apache web server application, and the remaining 5% to all other applications. The web server is unable to interfere with the L2 cache space of the database and vice-versa.

their target platform, they used a simulation environment (SimpleScalar) that does not take the interference of the operating system into account.

There have been other software-based approaches that explore mitigating interference in the shared cache among multiple applications, however they are unable to provide the stronger guarantee of isolation. These previous approaches include the pioneering symbiotic scheduling work done by Snavely et al. [Snavely and Tullsen 2000; Snavely et al. 2002], the work by Fedorova et al. to compensate for the interference after it has already occurred [Fedorova et al. 2007], and the work by Zhang et al. to reduce the impact of interfering threads by using duty-cycle modulation and cache prefetcher adjustment techniques [Zhang et al. 2009b].

4.3 Design of Cache Partitioning Mechanism

We first describe space partitioning the cache in general, followed by how to perform partitioning using a purely software-based approach at the operating system level.

4.3.1 Space Partitioning the Cache

Our approach in having the operating system manage the shared cache among the cores is to space-partition its capacity: A large shared cache is divided into smaller private caches, each dedicated to a set of applications. For example, in Figure 4.1, 75% of a shared cache is allocated exclusively to the MySQL database application, 20% is allocated exclusively to the Apache web server application, and the remaining 5% is allocated to all other applications. The web server is then not able to interfere with the cache space of the database and vice-versa.

Using a space-partitioning approach, the operating system can control the amount of cache space occupied by specific sets of applications and prevent them from interfering with each other. This kind of control allows the operating system to eliminate the disadvantage of space contention found in shared caches and obtain the advantage of space isolation found in private caches. Meanwhile, the advantages of a shared cache remain, such as faster inter-core communication, and the elimination of cache line replication across private caches. The disadvantage of space-partitioning is that applications are now restricted to a smaller portion of the cache, potentially suffering performance penalties and so the operating system should strive to minimize these occurrences.

Although one may intuitively expect that one should allocate a partition to a particular core, in actuality, each partition should be allocated to a particular set of applications (or a particular application). The problem of a shared cache is not that multiple *cores* could access the shared cache at a fine time granularity, but that multiple *applications* running on multiple cores could access the shared cache at a fine time granularity. As shown in Figure 4.1, the MySQL database application occupies both cores and can access its designated partition from both cores.

By allocating cache space on a per-application basis rather than on a per-core basis, two operating system management issues are beneficially decoupled by our approach, preventing further complexity from being added to the pre-existing operating system scheduling mechanisms and policies. First, within a single chip, the application can execute threads on any core and can migrate these threads among different cores without requiring further cache management actions. Second, multiple threads of a single application can execute on several cores at once without requiring further cache management actions. In Figure 4.1, the database and web server applications would be free to execute on any core or occupy as many cores as allowed for by existing operating system scheduling policies.

An important operating system management feature is to be able to dynamically resize the partitions while applications are executing, in order to adapt to changing requirements, such as when new applications are launched or current application resource requirements change. If an application changes phases and no longer needs as much cache space, the excess capacity can be allocated to other applications that would benefit.

Although the general idea presented here is described at the operating system level, our solution applies equally well to the virtual machine monitor level, where multiple operating systems can run simultaneously on a single physical machine. Each operating system can be viewed as a user-level application, and the virtual machine monitor can be viewed as the traditional operating system in the previous descriptions. That is, the virtual machine monitor can allocate portions of the shared cache for exclusive use by an operating system, restoring isolation properties to the system.

4.3.2 Space Partitioning the Cache by Software

To provide software-based L2 cache partitioning, we apply the classic technique of page-coloring [Bershad et al. 1994; Cho and Jin 2006 2007; Kessler and Hill 1992; Liedtke et al. 1997; Lynch et al.

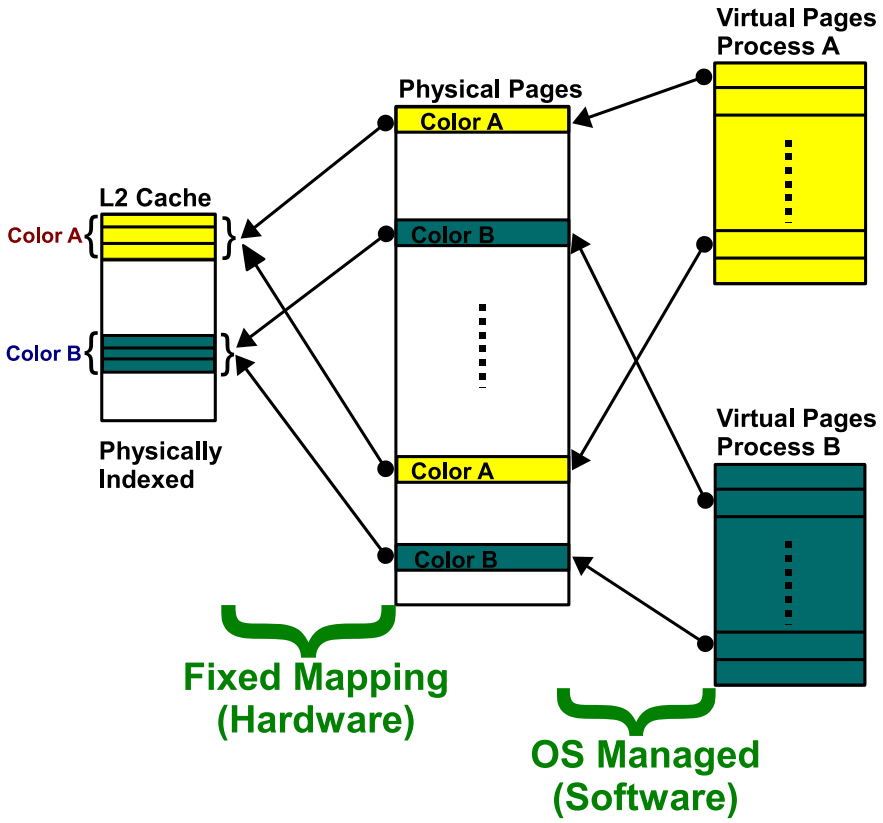


Figure 4.2: Page and cache section mapping. Due to the mapping of virtual pages to physical pages by the operating system, `Process A` has been restricted to exclusive use of the top portion of the shared L2 cache while `Process B` has been restricted to exclusive use of the bottom portion.

1992; Mueller 1995; Sherwood et al. 1999; Wolfe 1993]. In particular, Liedtke et al. were the first researchers to apply page-coloring at the operating system level to enable software-based cache partitioning [Liedtke et al. 1997]. However, their target was a real-time operating system running on a traditional single-core processor.

Our solution is based on two observations. First, in a physically-addressed cache, which is common in today's microprocessors, each cache line can hold data of only a subset of physical pages, and data of a physical page is held in a specific subsection of the cache. In other words, the content of a physical page will only be cached in a specific sub-area of the cache[2]. This mapping between physical page and cache section is defined by the hardware, as shown on the left side of Figure 4.2. For example, on an IBM POWER5 processor, each physical page maps directly onto one of 16 sections in the cache.

The second observation is that the operating system controls the mapping between virtual memory pages and physical memory pages. Hence the operating system can partition the cache between applications by partitioning physical memory between applications. As a result, each virtual page of an application can be mapped to a specified portion of the cache. When a new

[2]For processors with a virtually addressed L2 cache, our approach will not work.

physical page is required by an application, the operating system allocates a page that maps onto a section of the L2 cache assigned to that application. By doing so for every new physical page requested by the application, we isolate its L2 cache usage.

Figure 4.2 shows that there are several physical pages labeled `Color A` that all map to the same group of L2 cache lines labeled `Color A`. The figure also shows that physical pages of the same color are given to the same application. For example, physical pages of `Color A` have been assigned solely to application process `A`.

For non-targeted applications, the operating system can allocate either (1) a free physical page belonging to a designated miscellaneous partition so that all non-targeted applications share a partition, or (2) any free physical page so that non-targeted applications ignore all partition restrictions.

Other combinations are also possible with this mechanism, such as creating some private partitions and some shared partitions.

To dynamically resize an application's partition, either additional colors are assigned for partition growth, or some existing colors are revoked for partition shrinkage. For partition growth, this may require physical pages of one color to be reclaimed from one application and given to another. As a simplified example based on Figure 4.2, suppose that `Process B` terminates and the operating system wants to expand the partition size of `Process A` to also occupy `Color B`. To make immediate use of the newly available page colors, some of the existing pages of `Color A` would have to be moved to physical pages of `Color B`. That is, the contents of a physical page belonging to one color must be copied to a physical page belonging to another color. The corresponding mapping from the virtual-to-physical page must then be updated appropriately to point to the new physical page. Finally, the old page is freed.

The fixed mapping of physical page to cache section, as shown in Figure 4.2, is caused by the hardware's interpretation of bits shown in Figure 4.3. The upper n bits of the L2 cache set number field overlap with the bottom n bits of the physical page number field. Since the operating system has direct control of the physical page number field, it has n bits of influence on the L2 cache set number. On the POWER5 processor, n has the value of 4, resulting in 16 distinct page colors. Up to 16 partitions are possible when a distinct color is designated exclusively to each partition. The lower m bits of the L2 cache set number, which are beyond the direct control of the operating system, means that there are 2^m physically contiguous cache sets per page color.

Note that there are no bits in the physical address that are related to set-associativity because eviction within each set is managed at run-time by the hardware using a least-recently-used (LRU) policy[3].

The L2 cache on the POWER5 is physically implemented using 3 smaller caches of 640 kB each,

[3]Due to the high costs of implementing the LRU policy in silicon, processors typically implement a pseudo-LRU policy, which closely approximates true LRU [Al-Zoubi et al. 2004]. For example, the POWER5 processor implements a pseudo-LRU policy known as pairwise-compare [Zhang et al. 2008].

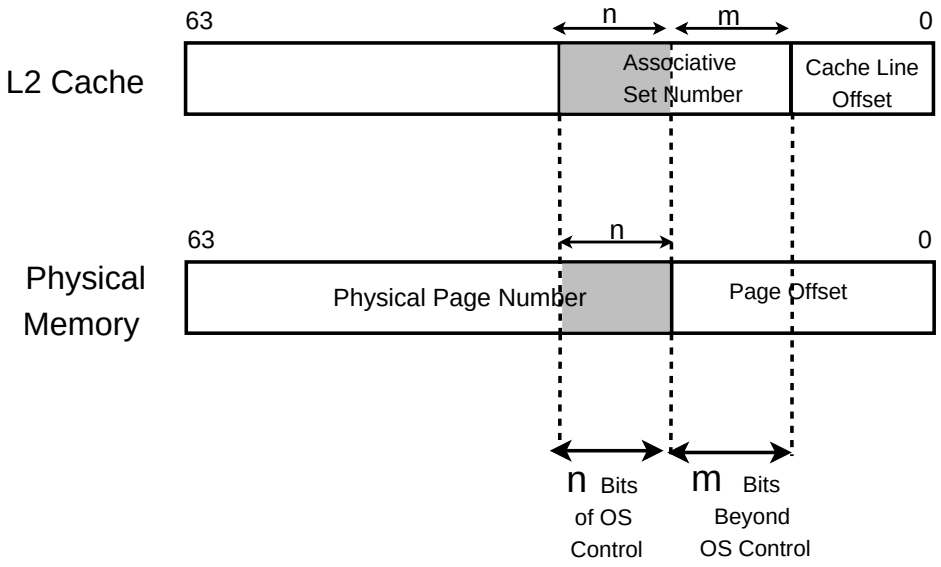


Figure 4.3: Bit-field perspective of mapping from physical page to cache section. Since the operating system has direct control of the physical page number field, it has n bits of influence on the L2 cache set number, resulting in 2^n distinct page colors.

each known as a cache *slice*. The slice which a physical address is mapped onto is determined by a hardware-implemented hash function using physical address bits 8 to 27 inclusively. Unfortunately, 4 of these bits are beyond the direct control of the operating system, meaning that slice usage appears largely uniformly random. Having direct control of the L2 slice usage would have enabled us to support 48 partitions.

Our L2 cache partitioning mechanism also causes the L3 victim cache of the POWER5 to be divided into 16 partitions. The derivation is similar to the L2 cache derivation and is therefore not shown. Each partition in the L2 has a direct mapping to a corresponding partition in the L3 victim cache.

4.4 Implementation of Cache Partitioning Mechanism

We implemented our cache management solution in Linux 2.6.15 by modifying the physical page allocation component of the operating system. For the dynamic partition resizing capability, the virtual-to-physical page table management code was modified.

4.4.1 Page Allocation Within Cache Sections

In standard Linux, each *processor*[4] has a private list of free physical pages. When an application needs to allocate a new physical page, one is dequeued from the local processor's free list. By default, this per-processor free list does not differentiate between different page colors.

Although partitioning physical memory is a fairly simple concept, its implementation in the

[4]Linux uses the term *processor* to generically refer to any execution unit, such as a hardware thread or a core.

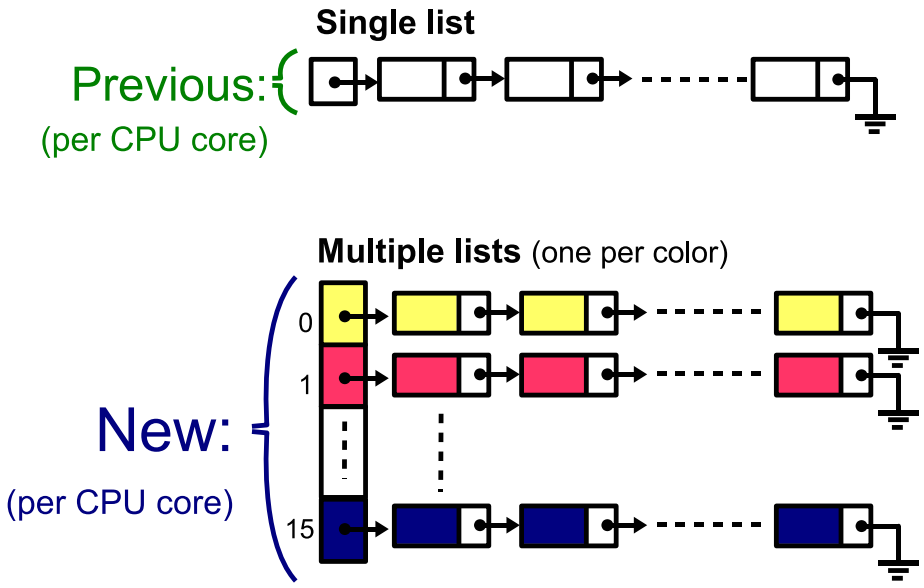


Figure 4.4: Implementing cache partitioning in Linux by modifying its physical page free list. The single free list is converted to multiple free lists, one per page color.

Linux kernel must be done carefully to prevent any negative performance side effects. In our first attempt, we simply used a single free list of physical pages for each processor detected by Linux. Having a single free list, however, incurred frequent and expensive linear searches of the potentially long free list in order to find a physical page of the suitable color.

Another problem with having a single free list is that upon application termination, a large number of pages are freed and put at the head of the free list. Since these pages were assigned to the recently terminated application, they may not be suitable for another application. As a result, a linear traversal of the free list scans through a potentially large number of unsuitable pages before it finds the first suitable page.

To address this issue, we converted the single free list into multiple free lists (still on a per processor basis), as shown in Figure 4.4. Since the POWER5 L2 cache can be divided by software to have a maximum of 16 colors, we had 16 free lists for each processor in the system. Each list contains free physical pages that map to a designated section of the L2 cache. Having multiple free lists, each corresponding to a distinct page color, dramatically accelerated the process of finding a suitable page, as it removes the need for linearly searching the free list.

When an application requires a physical page, the operating system determines which colors are eligible to the application and only selects a page from the appropriate free list. A simple Round-Robin scheme is used when multiple free lists are eligible in which to select a free page.

When a large number of physical pages are requested at once, Linux can allocate a group of physically contiguous pages using its global "buddy allocator" as long as the groups are powers of 2 in size. To support this, the Linux global buddy allocator maintains lists of groups of physically contiguous free pages of size 1 to 1024 pages (i.e., level 0 to level 10). For page allocation of

levels higher than 0 we use the single free list of the target level (i.e., for levels higher than 0, we do not use separate lists per partition). We traverse the list to find a suitable page group that maps to the target L2 partition. Note that the suitable page group, due to its physically contiguous nature, will not consist of physical pages of solely 1 color but of several adjacent colors. Consequently, for partition sizes of 2 adjacent colors or greater, these higher-level page allocations requests (contiguous physical page allocation) can be satisfied. Since allocations at higher levels is fairly rare, we do not foresee this case impacting performance significantly.

When a new physical page is being allocated for an application and its assigned partition free lists are empty, then the local, per-processor allocator must request additional free pages from the global Linux buddy allocator. Due to limitations in our prototype implementation, a problem may arise in that it is possible that none of the pages returned by the buddy allocator have the color of the target partition. Even repeated attempts may be unsuccessful. We employ a configuration parameter, `MaxTry`, to limit the number of such attempts. If, after `MaxTry` attempts, the partition free list remains empty, the physical page is allocated from another partition free list chosen randomly. The default value for `MaxTry` is set to 100[5].

There can be scenarios when there are no physical pages available of a suitable color for the application, despite physical pages of other colors that are available. Under such a scenario, instead of forcing the use of disk swap-space, which has orders of magnitude higher access latency than main memory, the previously described `MaxTry` policy randomly selects any available physical page. Of course, these actions can re-introduce cache space interference, but main memory access latencies are preferable to disk swap-space latencies. Subsequent techniques such as hot-page coloring, by Zhang et al., could be used to reduce any potential cache interference cause by occupying other colors [Zhang et al. 2009a]. Their hot-page coloring technique could be used inversely to identify cold pages of the application. These cold pages could then be moved to occupy other available colors without causing much cache interference to other applications due to their low frequency of access, and thus making suitably-colored pages available once again.

4.4.2 Dynamic Partition Resizing

Dynamic partition resizing means to either allocate additional colors to an application for partition growth, or reclaim some existing colors from an application for partition shrinkage, all while the application is running. In either case, it is necessary to *move* some existing application pages from old colors to new colors. To affect a move, the content of a physical page of an old color must be copied to the physical page of a new color. When a partition grows, a suitable number of its pages belonging to old colors are *moved* to the newly available colors. In contrast, when a partition shrinks, all pages belonging to invalid colors are *moved* to the remaining valid colors.

To implement page moves, the application's page table, which contains the mappings from

[5]The `MaxTry` limitation applies to our static partitioning experiments but is fixed (solved) in our dynamic partitioning experiments, described in Section 4.6.4.

virtual-to-physical page, must be traversed to identify the affected pages and each affected physical page must be copied to a new target page. More precisely, for each eligible page table entry: (i) the page table entry is locked, (ii) the contents of the entry and corresponding TLB entry, if present, are cleared so that any subsequent accesses to the corresponding virtual memory page by the application will trigger a page fault, pausing the application, (iii) the physical page is copied, (iv) the appropriate meta-data are adjusted, and (v) the page table entry lock is released. If the application was paused because it was attempting to simultaneously access the target virtual memory page, it is resumed, causing it to automatically access the new physical page instead of the old one.

In addition to performing the copying of contents from the old page to the new page, a number of housekeeping tasks may need to be done. First, meta-data from the old physical page must be copied to the new physical page, such as page permissions, dirty bit status, and reference counts. Second, for file-mapped pages, any references to the old page in the page caches are updated to point to the new page. Third, references to the old page in the page-eviction least-recently-used (LRU) lists are updated to point to the new page.

This partition resizing is done while the application is running. At any time during the partition resizing period, only one page at a time is ever out of commission. This approach allows the application, if it is multithreaded and simultaneously running on other cores or hardware threads, to continue to simultaneously access all other pages except for the currently targeted page. A thread of the application will only be briefly paused if it attempts to access the same target page during the exclusive period, and resumed once the exclusive period has ended.

4.5 Experimental Setup

The multiprocessor used in our experiments is an IBM OpenPower 720 computer, as specified in Table 4.1. It is an *8-way* POWER5 consisting of a 2×2×2 SMP×CMP×SMT configuration[6]. Each chip has 1.875 MB of shared L2 cache that is shared between the on-chip cores. There is an off-chip 36 MB L3 victim cache per chip. As mentioned previously, Linux 2.6.15 was used and modified to allow for L2 cache partitioning. Our modifications consist of approximately 500 lines of code (LOC), stemming from 250 LOC for the static partitioning mechanism and 250 LOC for the dynamic partitioning mechanism. With the given hardware, 16 distinct page colors are possible, allowing for a maximum of 16 partitions, each of size 120 kB in the L2 cache and 2.25 MB in the L3 victim cache.

To create a controlled execution environment for our experiments, the Linux scheduler was modified to disable the default reactive and pro-active task migration mechanisms and policies. Our partitioning mechanism has no compatibility issues with process migration across cores since physical-to-virtual page mappings remain unchanged. In addition, and for the same reasons, our

[6] 2 chips × 2 cores per chip × 2 hardware threads per core.

Item	Specification
# of Chips	2
# of Cores	2 per chip
CPU Cores	IBM POWER5, 1.5 GHz, 2-way SMT
L1 ICache	64 kB, 128-byte lines, 2-way set-associative, per core
L1 DCache	32 kB, 128-byte lines, 4-way set-associative, per core
L2 Cache	1.875 MB, 128-byte lines, 10-way set-associative, 14 cycle latency, per chip
L3 Victim Cache	36 MB, 256-byte lines, 12-way set-associative, 90 cycle latency, per chip, off-chip
RAM	8 GB (2 banks × 4 GB), 280/310 cycle latency local/remote

Table 4.1: IBM OpenPower 720 specifications.

mechanism is independent of the number of cores sharing the L2 cache. In our experiments, we use only 1 chip of the system.

The workloads used were SPECjbb2000 and SPECcpu2000. The IBM J2SE 5.0 JVM was used to run SPECjbb, under a 1 warehouse configuration. For SPECcpu, 20 out of the 26 applications were run using the standard *reference* input set. The remaining 6 applications, which were mostly Fortran-based, did not compile successfully. Although we did not intentionally leave out those applications to bias the results in our favour, we recognize that there is the possibility of unintentional biasing because we are ignoring a representative set of Fortran-based workload characteristics. To simulate a multiprogrammed server environment, various combinations of these applications were run together.

4.6 Results

4.6.1 Impact of Partitioning

With software-based partitioning, we have the ability to easily study the impact of L2 cache size on execution time. Figure 4.5 shows the impact of varying the L2 cache size using our partitioning mechanism. Each application was run alone on a single core. Each point is the average of 3 runs, and the error bars indicate the minimum and maximum values seen. The machine was rebooted before each run, leading to the same initial system state before each run. Applications `gap`, `wupwise`, `mesa`, `gcc`, and `sixtrack` are not shown because they had flat curves similar to `mgrid` in Figure 4.5p. The results of Figure 4.5 are similar in spirit to the initial graphs shown by Qureshi and Patt but our results come from a software implementation running on a real system [Qureshi and Patt 2006]. We show application-reported run times for the entire run of the application, which includes all overheads. Our results here are difficult to compare against those obtained by Qureshi and Patt because we use a 1.875 MB L2 cache that is partitioned at physical page granularity while Qureshi and Patt used a 1 MB L2 cache partitioned by hardware at cache line, set-associative way granularity. Furthermore, they generated their results by simulating a fairly short fraction of application execution of approximately 250 million instructions, whereas we ran applications for

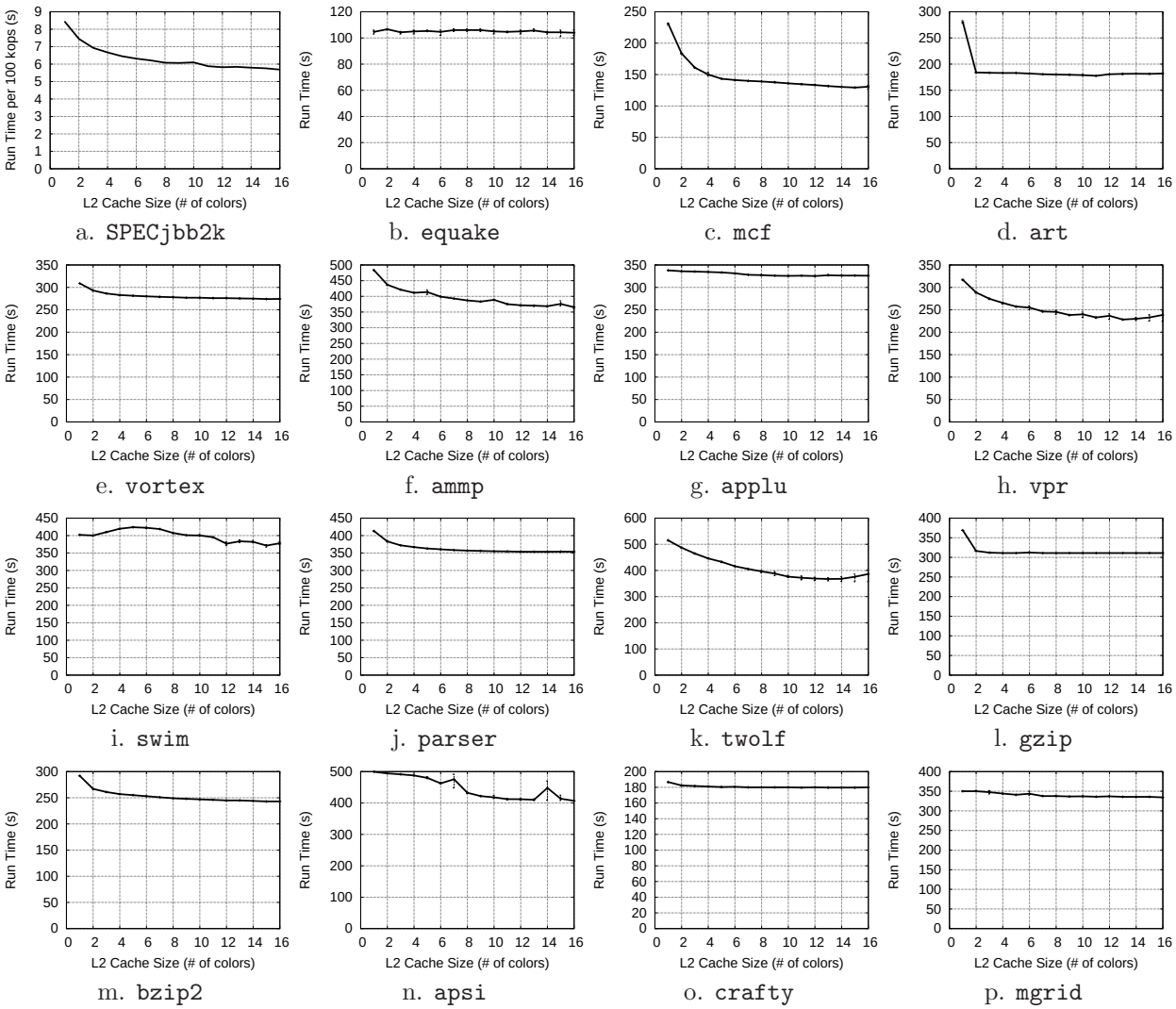


Figure 4.5: Single-programmed application performance as a function of L2 cache size. The performance impact is shown in terms of total application execution time. In general, as the L2 cache size is increased, execution time monotonically decreases.

much longer periods of several tens of billions of processor cycles.

Our results indicate that for some applications, having a small fraction of the cache is sufficient to achieve performance close to the performance achieved with the entire cache. For example, SPECjbb requires 8 colors, `mcf` requires 4 colors, and `art` requires 2 colors.

Most graphs show monotonically decreasing execution times as the cache size is increased, as expected. However, there are a few exceptions. For instance, `swim` shows increasing execution times as the cache size is increased from 1 to 5 colors, `twolf` shows increasing execution times as the cache size is increased from 12 to 16 colors, `apsi` shows anomalies at 7 and 14 colors (multiples of 7), and `ammp` shows anomalies at 5, 10, and 15 colors (multiples of 5). For `swim`, this anomaly is likely due to the fact that it may be exceeding the `MaxTry` threshold, enabling it to use sections of the L2 cache that were not assigned to it and thus escaping the imposed partition size restrictions. For other applications, we believe that these anomalies are due to two factors. First, each possible cache

partition size configuration leads to different virtual-to-physical page mappings, which may cause pressure on various cache sets of the set-associative L2 cache. Second, since cache partitioning was applied only to applications and not the operating system itself, interference from the operating system may result. For example, there may be the presence or absence of cache interference between the application and operating system meta-data, such as the virtual-to-physical page table entries. Such interference may potentially lead to slower resolution of TLB (translation look-aside buffer) misses when the page table entries are not present in the L2 cache and must be obtained from main memory, as illustrated by Soares et al. [Soares et al. 2008]. Finally, these anomalies cannot be due to altered cache miss patterns stemming from physical memory reclamation performed by the operating system, since the machine was rebooted before each run to create the same initial system system before each run.

It is important to note that with the initial single-programmed (single application) results shown in Figure 4.5, the impact of co-scheduling two or more applications on a single chip without software-based cache partitioning cannot be easily predicted. One important characteristic that is missing from Figure 4.5 is the L2 cache usage demands of each application. For example, an application could exhibit streaming behaviour consisting of high L2 cache access frequency and no reuse frequency, leading to a high miss frequency no matter how much L2 cache is allocated exclusively to the application.

Figure 4.6 shows the impact of software cache partitioning on performance for seven combinations of multiprogrammed workloads. Each application was run on its own core but within the same chip so that the L2 cache is shared[7]. The units shown are average IPC (instructions-per-cycle) improvement per billion processor cycles as reported by the hardware PMU tools developed by our research group [Azimi et al. 2005]. The performance is normalized to the performance of the same combination of applications without partitioning, also known as the uncontrolled sharing configuration. Each multiprogrammed workload was run 15 times in order to obtain 15 pairs of points for each graph, corresponding to the 15 possible pairs of partition sizes.

The bottom x-axis shows the number of colors (N) given to one application, while the remaining $16 - N$ colors are given to the second application, as indicated by the top x-axis. Note that the 2 x-axes run in opposite directions so that a vertical line drawn at any point will indicate a total of 16 colors allocated among the two applications, meaning that the entire L2 cache is used. The y-axis indicates performance, in terms of instructions-per-second (IPC), of an application, normalized to the performance where both applications are running simultaneously but with uncontrolled sharing of the L2 cache. The y-axis performance values are the averages over the first 60 billion cycles of execution after ignoring an initial warmup period of 30 billion cycles, captured using a hardware PMU window size of 1 billion cycles. A vertical line drawn at any point in the graph indicates the normalized performance of the two applications for a given partitioning of the L2 cache. For

[7]Examining the impact of SMT, by running both applications on the same core, is beyond the scope of our work.

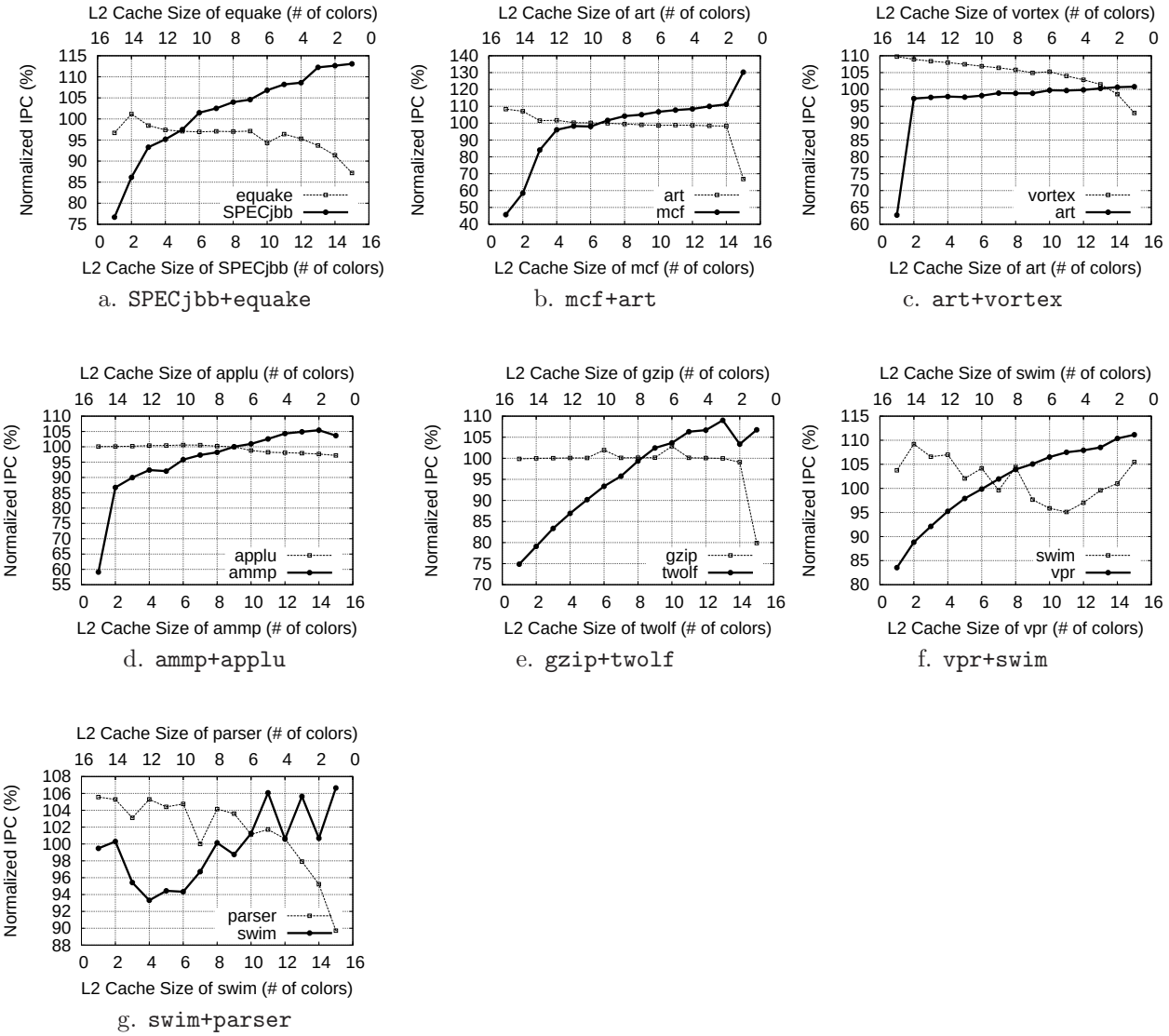


Figure 4.6: Multiprogrammed workload performance as a function of L2 cache size. The performance is normalized to the performance where both applications are running simultaneously (on the same chip, one application per core) but with uncontrolled sharing of the L2 cache. The bottom x-axis shows the number of colors (N) given to one application, while the remaining $16-N$ colors are given to the second application, as indicated by the top x-axis. A vertical line drawn at any point indicates the normalized performance of each of the two applications for a given partitioning of the L2 cache. Cache partitioning enables (1) performance benefits to be extracted from one application without significantly affecting the other, or (2) a selectable trade-off of performance between applications.

example, when SPECjbb is given 12 colors in Figure 4.6a, `equake` is given the remaining 4 colors. The graph indicates that SPECjbb can achieve a throughput improvement of up to 8% (12 colors) while `equake` is penalized by less than 5% (4 colors). If SPECjbb is intended to be the high-priority application while `equake` is the low-priority application, then these priorities could be enforced with software-based cache partitioning. As an extreme example, SPECjbb could be given 14 colors with the remaining 2 colors given to `equake`, resulting in a 13% improvement to SPECjbb while penalizing `equake` by 8%.

For the SPECjbb+`equake` combination, we used a `MaxTry` value of 25,000, rather than the default. Using a lower value caused the performance of SPECjbb to begin showing degradations from 11 to 15 colors. This occurred because `equake` would exceed the `MaxTry` 100 threshold frequently since it was allowed only 1 to 5 colors. Upon this occurrence, `equake` would obtain a physical page belonging to SPECjbb instead.

Figure 4.6b indicates that the performance of `mcf` can be improved by up to 11% (14 colors) without noticeably affecting `art`. In Figure 4.6c, `vortex` can be improved by 5% (6 colors) without affecting `art`. If `art` is a lower-priority task, then `vortex` can be improved by up to 8% (14 colors) while penalizing `art` by 3%. In Figure 4.6d, `ammp` can be improved by 5% (14 colors) while penalizing `applu` by 2.5%. In Figure 4.6e, `twolf` can be improved by 8% (13 colors) without penalizing `gzip`. The drop in IPC for `twolf` at 14 and 15 colors is likely due to the interference from `gzip` upon `gzip` exceeding the `MaxTry` 100 threshold. A similar situation, in which the `MaxTry` threshold was exceeded, likely occurred in Figure 4.6f as well, from 4 colors to 1 color in `swim`. Exceeding the `MaxTry` threshold would enable `swim` to use sections of the L2 cache belonging to `vpr`. Fortunately, this situation helped `swim` without significantly affecting `vpr`. An aggregate peak performance of 17% was achieved stemming from an 11% improvement to `vpr` (15 colors) and 6% improvement to `swim` (1 color). Finally, Figure 4.6g also shows the same `MaxTry` threshold phenomenon in `swim` between 1 to 4 colors.

In general, cache partitioning can either: (1) allow one application to increase performance without significantly affecting the other, or (2) enable a trade-off spectrum where the performance of one application, perhaps a low-priority one, can be sacrificed for increase performance of another application, perhaps a higher-priority one.

Although not shown here, we observed no impact on the L1 instruction cache in the SPECcpu applications. However for SPECjbb, as the size of the L2 cache was decreased from 5 colors to 1 color, we observed a noticeable increase in the instruction retirement stall rate due to L1 instruction cache misses.

4.6.2 Analysis of Interference

Understanding and characterizing the performance impact of sharing resources on a multicore processor is an essential part of (1) predicting which combinations of applications exhibit performance interference and, (2) quantifying the potential performance improvements of controlling resource

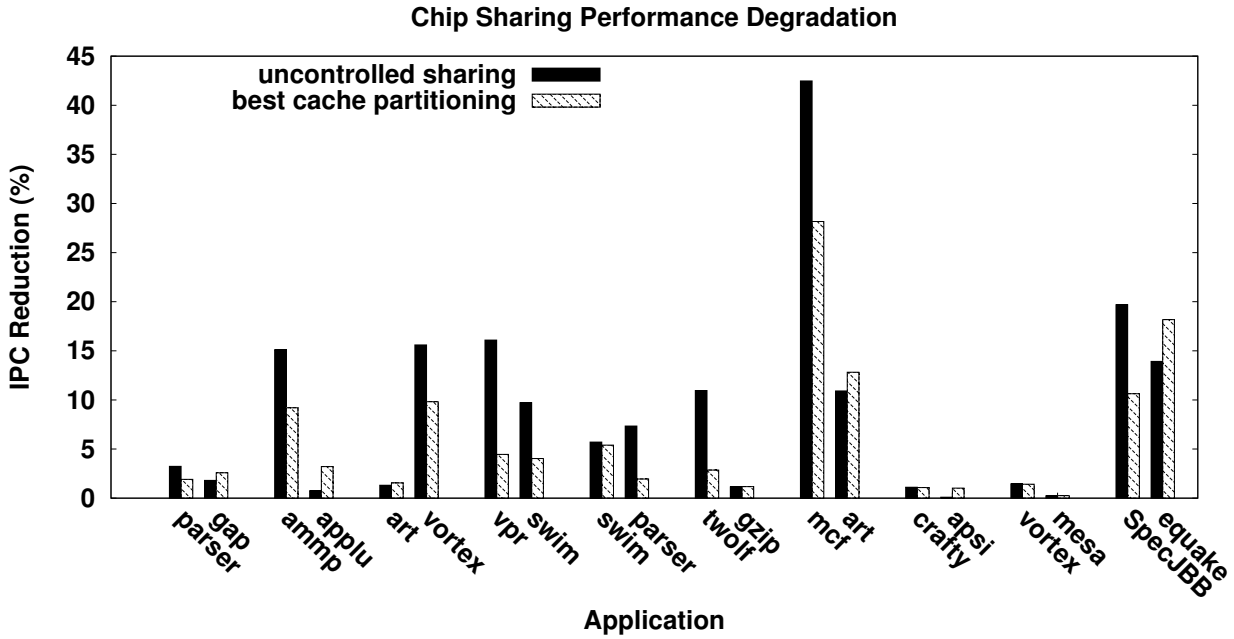


Figure 4.7: Performance comparison of applications executing in pairs sharing the chip, with and without isolation. The y-axis is normalized to the performance of the application executing alone on the chip. Lower y-axis values are better. Cache partitioning is able to significantly reduce the IPC degradation, due to L2 cache sharing, for one of the applications while possibly slightly degrading the IPC of the second application.

sharing, in this case, L2 cache sharing. In this section, we demonstrate that cache partitioning can recover up to 70% of the IPC degraded due to chip sharing. Furthermore, we detail how hardware PMUs can be used to predict the potential performance interference between applications executing on different cores of the same chip.

Figure 4.7 shows the performance degradation suffered by applications when executing as a pair, compared to when running alone (single-application mode). Each application in the multi-programmed pair is executed on its own core but within the same chip, thus sharing the L2 cache. Two different setups are plotted, normalized to single-application mode: (1) the reduction in IPC of applications executing with no cache partitioning (uncontrolled sharing), and (2) the reduction in IPC of applications executing using a fixed (static) partition size that was empirically found to be the "best" (best cache partitioning) for the multiprogrammed pair.

With most combinations shown, cache partitioning is able to significantly reduce the IPC degradation, due to chip sharing, for one of the applications while possibly slightly degrading the IPC of the second application. The worst degradation seen is in `equake` when run together with SPECjbb; the IPC of `equake` suffers a 4% decrease, while enabling a 9% improvement in the IPC of SPECjbb. The best improvement is seen in `twolf` when run together with `gzip`. This combination shows that cache partitioning can recover up to 70% of degraded IPC due to chip sharing.

The main reason behind most of the benefit seen by controlling L2 cache sharing is the fact that while some applications are memory intensive in their behaviour, they may not benefit from using the entire L2 cache. This is the case, for example, for `art`. Figure 4.8 shows the number of processor

cycles in which instruction retirement is stalled due to L1 data cache misses in a billion processor cycles, with varying L2 cache partition sizes, as collected by the POWER5 hardware PMU. In this figure, each application is executed alone on the chip. Figure 4.8a shows the variation in memory access related stalls for `art`. There are two notable observations. First, the run time curve shown in Figure 4.5d closely resembles the curve in Figure 4.8a, demonstrating that for this application, memory stalls seen by the core are strongly related to its performance. Second, it is clear that giving more than 2 colors to `art` does not improve its memory performance.

The curves for `mcf` and `vortex` in Figure 4.8, however, show a different behaviour. The number of memory related stalls monotonically decreases as the number of colors grows. As can be seen in Figure 4.7, both `mcf` and `vortex` show performance benefits with cache partitioning when executing along side `art`, because the partitioning isolates the lack of locality seen in `art` and avoids the replacement of useful cache lines belonging to the other application, which otherwise would have been the case with the LRU (least-recently-used) hardware mechanism for cache line replacement[8].

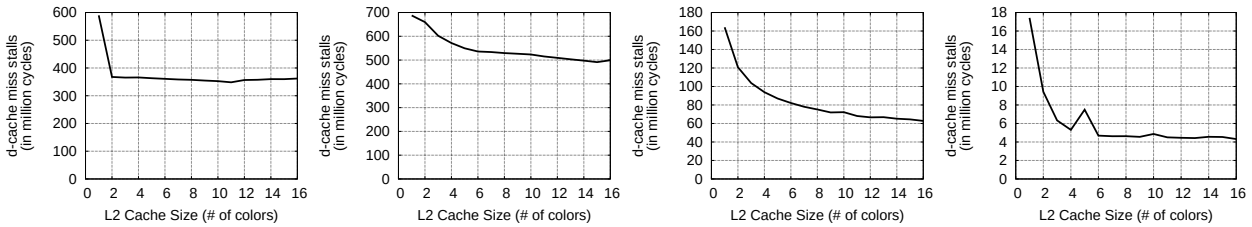
Finally, the performance of a few application combinations are not affected when sharing the same chip. This is the case for the `gap+parser`, `apsi+crafty`, and `vortex+mesa` combinations shown in Figure 4.7. This can also be explained by analyzing the instruction retirement stalls due to L1 data cache misses. As can be seen in Figure 4.8d, although `mesa` shows sensitivity to varying cache partition sizes, instruction retirement is stalled due to L1 data cache misses for, typically, only around 5% of the processor cycles. This indicates that `mesa` has very low cache requirements and is unlikely to replace important cache lines from applications executing on a sibling core when using the default LRU hardware mechanism for cache line replacement.

4.6.3 Stall Rate Curve Versus Miss Rate Curve

Figure 4.9 shows the L2 miss rate curves for the same four applications shown in Figure 4.8. While the stall rate curves (SRCs) in Figure 4.8 directly measure instruction retirement stalls caused by the memory hierarchy, the L2 miss rate curves in Figure 4.9 measure only a single component of the performance picture, which is the *rate* of misses experienced at the L2 cache only.

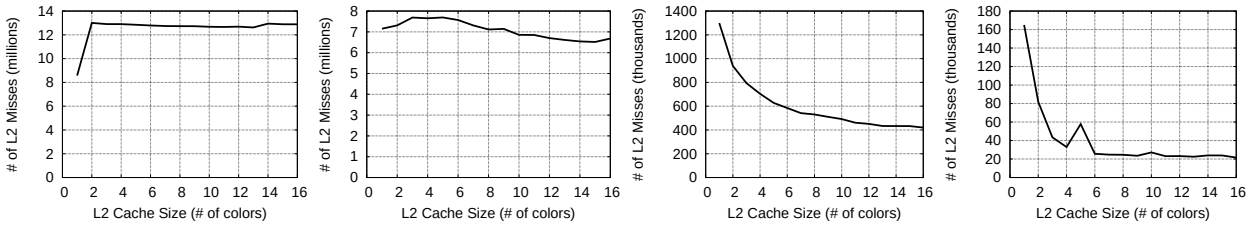
It is interesting to note that in some scenarios, the L2 miss rate is not sufficient to accurately predict the performance impact of memory operations because it does not account for the penalty of misses. In a multi-level cache hierarchy, the penalty of an L2 cache miss can vary dramatically depending on the source from which the cache miss is served. For example, when varying from 1 to 2 colors with `art`, although there is a significant performance increase (Figure 4.5d) and L1 data cache stalls drop (Figure 4.8a), the L2 miss rate curve does not show a corresponding decrease, as one might have expected (Figure 4.9a). Rather, it shows an increase in the miss rate. By examining the L3 victim cache and local memory hit rate curves, shown in Figure 4.10, we can see reason for

[8]Due to the high costs of implementing the LRU policy in silicon, processors typically implement a pseudo-LRU policy, which closely approximates true LRU [Al-Zoubi et al. 2004]. For example, the POWER5 processor implements a pseudo-LRU policy known as pairwise-compare [Zhang et al. 2008].



a. art b.mcf c. vortex d. mesa

Figure 4.8: The data cache stall rate curves (SRCs) shown here illustrate the sensitivity of the processor pipeline to stalls caused by data cache misses, as a function of L2 cache size. The y-axis indicates the number of processor cycles in which instruction retirement is stalled due to L1 data cache misses in a billion processor cycles. The data is obtained from the POWER5 hardware PMU. Single-programmed mode.



a. art b.mcf c. vortex d. mesa

Figure 4.9: The L2 miss rate curves (MRCs) shown here indicate the rate of L2 cache misses suffered as a function of L2 cache size. The y-axis indicates the number of L2 cache misses per billion processor cycles, as reported by the POWER5 hardware PMU. Single-programmed mode.

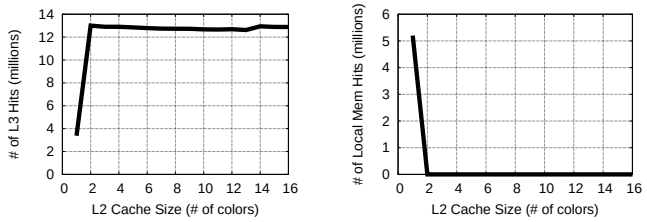


Figure 4.10: L3 victim cache and local memory hit rates for art, per billion cycles, as reported by the POWER5 hardware PMU. Single-programmed mode.

this anomaly in the L2 miss rate curve.

In the 1 color case, due to the simultaneous partitioning of the L2 and L3 caches (described in Section 4.3.2), the fast L3 cache is too small to successfully resolve most requests and so the slower local main memory must handle roughly half of them. That is, the L3 cache successfully services approximately 3.5 million accesses while the local main memory must handle 5 million accesses, per billion cycles. However, in the configurations with greater than 2 colors, nearly all L2 cache misses are successfully resolved in the fast L3 cache (approximately 13 million hits per billion cycles) and almost none reach the slower local main memory. Since L3 cache hits are significantly faster than local main memory accesses, there is a significant increase in application progress and hence, an increase in application cache access rates. It is this higher rate of application progress, at cache size configurations of 2 colors or greater, that leads to a higher number of hits and misses (per billion

cycles) in the L2 cache, as shown in Figure 4.9a.

Our experience with Figure 4.8 has led us to realize that a more encompassing metric for performance prediction and explanation is an application's rate of instruction retirement stall caused by memory latencies, as a function of L2 cache size. Using instruction retirement stall as a metric means that we are directly examining scenarios where a processor is not completing work in a timely manner. On our system, this is roughly equivalent to the instruction retirement stall rate caused by L1 data cache misses, as measured in Figure 4.8 directly with the hardware performance counters. This instruction retirement stall rate curve (SRC), caused by memory latencies, incorporates important factors such as (1) the L2 cache miss rate; (2) instruction retirement stall sensitivity to L2 cache misses; (3) non-uniform access latencies to lower levels of the memory hierarchy, such as the L3 cache, local main memory, and remote main memory; and (4) shared memory bus contention.

Measuring instruction retirement stalls due to L1 data cache misses will capture instances where an L1 data cache miss is not serviced by the L2 cache quickly enough, leading to extra stalls. Applications that have adequate memory-level parallelism may not be as sensitive to stalling on L2 caches misses because they are able to hide the latency of a portion of their L2 cache miss resolutions behind others that have caused the processor to stall [Qureshi et al. 2006]. The L1 data cache stall rate will indicate only detrimental cache misses. Non-uniform access latencies are captured by the L1 data cache stall metric when latencies are large enough to cause instruction retirement stalls. Memory bus contention exhibits similar impact on this metric.

4.6.4 Benefits of Dynamic Partition Resizing

In the dynamic partitioning experiments of this section (and the next section), the `MaxTry` threshold described in Section 4.4.1 was eliminated with further modifications to the Linux buddy allocator, enabling it to return physical pages of the desired color(s). These modifications involved: (1) maintaining multiple free lists within each level of the buddy allocator, (2) enabling each level to recursively ask the subsequent "higher-order" level for physical pages of the desired color(s) when there are insufficient amounts at the current level, and (3) enabling each level to service such requests from a preceding "lower-order" level.

In Figure 4.11, we show a scenario where dynamic partition resizing is beneficial. We use the SPECjbb2000 application server benchmark on our POWER5 system. During the first 2 minutes of execution, SPECjbb services 2 clients that are simultaneously making requests, each to its own SPECjbb warehouse. During this phase, a partition size of 1 color is adequate to meet a *service-level agreement* of 9400 operations-per-second on a per client basis. However during the second 2 minutes of execution, 4 clients are now making requests, each to its own separate SPECjbb warehouse. In order to continue satisfying the performance target, it is necessary to grow the partition size to 2 colors. In contrast, if the partition size had remained statically set to 1 color, then the application would have been 15% below the performance target, as indicated by the left-

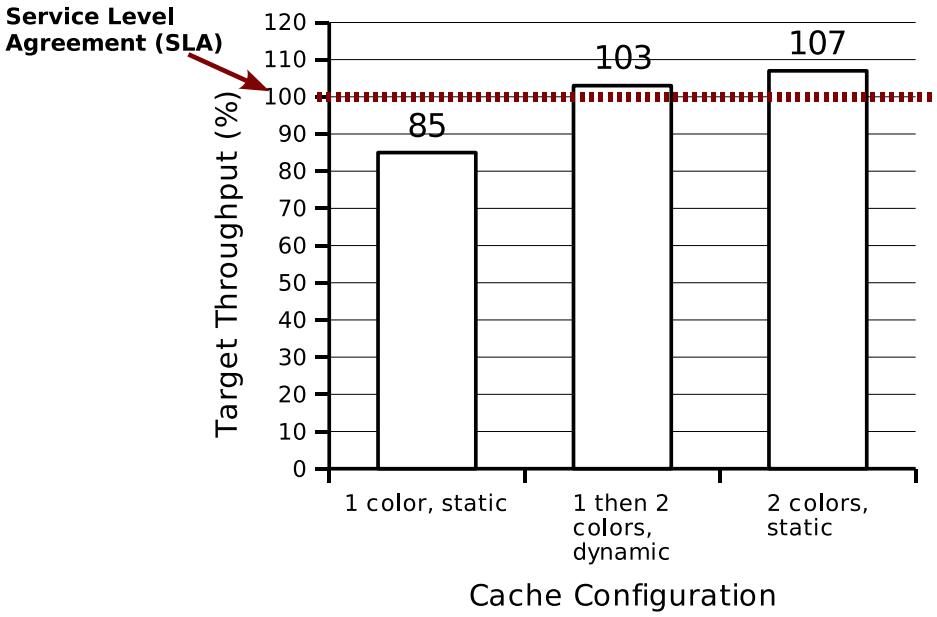


Figure 4.11: The effectiveness of dynamic partition resizing in matching workload variation. During the first 2 minutes, SPECjbb 2000 services 2 clients, while during the second 2 minutes, it services 4 clients. Compared to a statically chosen partition size, dynamic partition resizing can satisfy performance targets under workloads variations without requiring the over-commitment of cache space.

most bar in Figure 4.11. The middle bar shows that with dynamic partition resizing, SPECjbb is able to meet the performance target under workload variations. The right-most bar shows the performance of SPECjbb if 2 colors had been statically allocated. Although this static configuration would allow SPECjbb to meet performance targets, during periods of low workload demand, it would consume cache space that may have been better utilized by another application. Each data point is computed from the average of 3 runs. The machine was rebooted before each run, leading to the same initial system state before each run.

The benefit of dynamic partition resizing is two-fold. First, if application workload demand *increases*, more cache space could be obtained to meet the performance target. Second, if the application workload demand *decreases*, excess cache space could be given to some other application that uses it more effectively.

4.6.5 Costs of Dynamic Partition Resizing

In order to evaluate the overhead of dynamic partition resizing, which is primarily due to physical page copying, we conducted the following experiment. We ran SPECjbb2000 with 1 warehouse to occupy the first half of the L2 cache. Then after a period of time called *Copy Interval*, we force the application to abandon the first half of the cache and occupy the second half of the cache instead. After another copy interval, we force the application to reverse its cache occupation. We repeat this back-and-forth copying process, executed on the same core and same SMT hardware thread, until the application has terminated. We then compare throughput results to the configuration

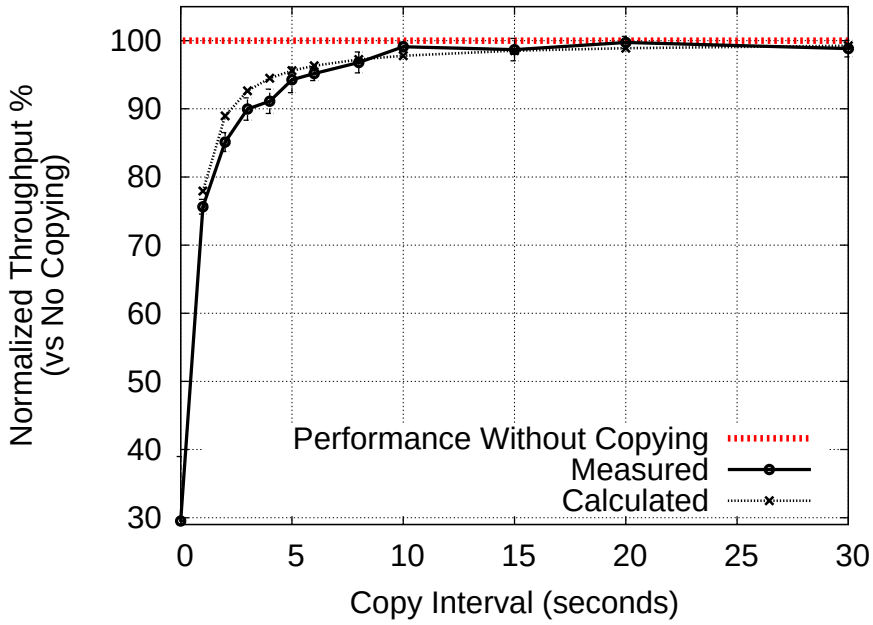


Figure 4.12: The overhead of partition resizing in SPECjbb 2000. 117 MB are copied each time. As the copy interval increases, the overall impact of page copying on performance diminishes because the cost is amortized over a longer period of time. In this particular setup of SPECjbb, dynamic partition resizing can effectively respond to phase changes as long as they are no more frequent than once every 10 seconds.

where the first half of the L2 cache is statically assigned to the application for the entire duration of execution.

By moving the entire application's cache occupation, we create a worst-case scenario in dynamic partition resizing that causes *all* physical pages belonging to old colors to be copied to new physical pages belonging to new colors (117 MB in this setup). In practice however, only a fraction of physical pages of the application need to be copied.

The curve labelled "Measured" in Figure 4.12 shows the results of this experiment. As the copy interval grows, the overall impact of page copying on throughput diminishes as the cost is amortized over a longer period of time. For SPECjbb in this particular setup, the impact of dynamic partition resizing is negligible as long as the copy interval is greater than 10 seconds. That means we can effectively respond to phase changes in SPECjbb as long as they are no more frequent than once every 10 seconds.

This pingpong effect means that approximately 117 MB are copied every X seconds. We measure the impact of this worst-case scenario by observing the application-reported average throughput number reported at the end of the workload. Each point is the average of 10 runs and the error bars indicate standard deviation. The machine was rebooted before each run, leading to the same initial system state before each run.

We also measured the time required to copy the contents from a 4096-byte old page to a new page. On average, 11,051 cycles, which is 7,368 ns, are required for an eligible page table entry to serviced, which includes the actual physical page copy and housekeeping work done before and after

on the meta-data. From this measured cost per page, the operating system can use this value, in combination with the number of pages that need to be migrated for an application, to calculate the total migration cost, which could help to determine whether a migration is worthwhile. Techniques to reduce the number of physical pages copied have subsequently been investigated, such as lazy page copying by Lin et al., and hot-page coloring by Zhang et al. [Lin et al. 2008; Zhang et al. 2009a]. As a sanity check, we used this measured cost per page to generate the curve labelled "Calculated" in Figure 4.12 and see that these calculated closely match the actual measured values.

In our implementation, we decided to leave alone physical pages that were being actively shared by multiple address spaces, such as shared pages containing shared library code. Examples of shared libraries include the standard C language, POSIX thread, and X Windows libraries. Shared pages that were not being shared at the time were copied. For SPECjbb, non-copyable pages made up less than 10% of the total pages. There are policy issues that would arise from moving these shared pages. For example, there may be a shared page that is used by both application `A` and `B`. If application `A`'s partition shrinks and the shared page must be moved, there are two possible locations in which to move the shared page: into application `A`'s partition or application `B`'s partition.

4.7 Discussion

Our software-based mechanism of cache partitioning may have an impact on future hardware design because it can facilitate the reduction in the degree of set-associativity in shared caches. Higher degrees of set-associativity is one possible hardware solution to mitigating cache line conflicts among applications. As a hypothetical example, a 16-way set-associative shared cache may experience the same miss rate as an 8-way set-associative cache that is partitioned using our software-based mechanism. Maintaining a low degree of set-associativity in the cache is a desirable hardware design goal because, in contrast, increasing set-associativity leads to a number of disadvantages, such as higher transistor counts, increased consumption of chip area, longer cache access latencies, and higher power consumption.

Although we have demonstrated cache partitioning on an IBM POWER5 processor, it is possible to partition the L2/L3 cache on other processors, such as the IBM PowerPC family and the Intel/AMD x86 family of processors [Lin et al. 2008; Soares et al. 2008; Zhang et al. 2009a]. Cache partitioning via the page coloring technique is possible on any processor that uses physical addresses, rather than virtual addresses, to access the cache. For example, shared off-chip L3 caches can be partitioned in this manner, as described in Section 4.3.2. In contrast, partitioning shared L1 caches is usually not possible because these caches are typically virtually addressed. In addition, given the relatively small size of L1 caches, partitioning them into smaller private caches may not be beneficial as there may be a severe increase in cache misses.

With current hardware indexing of cache lines, software cache partitioning is compatible with

larger page sizes up to a certain extent. As the size of a page grows by doubling its size, the number of distinct page colors and possible partitions in the L2 decreases by half. However, if the size of a page causes the number of distinct page colors and possible partitions to drop below two, then cache partitioning would no longer be possible.

Our partition mechanism does not create load imbalance on the main memory banks of the system since our POWER5 system make use of standard interleaved memory design. That is, the *words* of each page are mapped in a Round-Robin manner to the memory banks, so that the N^{th} word is located in memory bank N mod B, given B memory banks.

In this work, we have assumed that per application L2 MRCs and instruction retirement stall rate curves (SRCs), where stalls are caused by memory latencies, are available to the operating system as they are obtained during profiling runs and stored in a repository. In order to add a new application to the repository, these curves must be calculated by running the application (or at least a representative portion of it) several times (16 in our setup). Ideally, one might want to calculate an application's L2 MRC online with low overhead. Berg and Hagersten use a software approach based on data address *watchpoints* to calculate MRC online with a runtime overhead of 39% [Berg and Hagersten 2005]. In the next chapter, we demonstrate an improved technique to calculate the MRC in a lower overhead online manner, which does not cause 39% runtime overhead for the entire execution duration of the application.

We have also assumed that the L2 MRC and SRC of the application are stable throughout the execution of the application. In reality, each application goes through several *phases* that may have different memory access patterns. To react to such phase changes, dynamic repartitioning of the L2 cache is required, which may potentially incur significant costs in copying of data from one color to another, as evaluated in Section 4.6.5. However, if program phases are long enough to offset this overhead, then our software-based approach is still applicable. We will exam phase lengths in detail in the next chapter.

4.8 Concluding Remarks

In this chapter, we have demonstrated the benefits of the shared-cache management principle of providing isolation via a software-based cache partitioning mechanism and shown the potential gains in a multiprogrammed computing environment. Our mechanism allows for flexible management of the shared L2 cache resource.

Although we have implemented this mechanism at the operating system level to provide L2 cache space isolation between applications, it can also be applied at the virtual machine monitor level to provide L2 cache space isolation between guest operating systems. In addition, with the appropriate co-ordination and interface between the guest operating system and virtual machine monitor, cache partitioning for both purposes could be simultaneously and hierarchically performed.

Now that this mechanism is in place, in the next chapter we will extend this mechanism to

answer the next challenge of how to determine the optimal partition size for an application in a low-overhead online manner. With this challenge met, we will have developed the underlying technologies required to create a continuous optimization system that can (1) dynamically determine the optimal partition size for an application in an automated, online, low-overhead manner on existing multicore processors, and that can (2) dynamically adjust the size of the partition given to an application in an online, low-overhead manner.

Chapter 5

Provisioning the Shared Cache

"Measure twice, cut once." – Unknown

[1]An implicit requirement of the shared-cache management principle of providing cache space isolation is to provide a specified amount of cache space exclusively to an application. When partitioning the shared cache, the operating system must decide upon an appropriate size for each partition. One possible solution is to use a trial-and-error technique of simply trying several partition sizes and monitoring the resulting application performance. Another solution is to take a more analytical approach by using the cache miss rate curve of the application to determine an appropriate size. Such a curve reveals the trade-off spectrum between provisioned cache size and the resulting cache miss rate. However, obtaining the miss rate curve of a processor cache in an online manner is a challenging problem on existing processors.

In this chapter, we demonstrate that the hardware performance monitoring unit (PMU) and its associated hardware performance counters found in current commodity processors can be used to obtain a low-overhead, online approximation of the miss rate curve of the on-chip shared cache. On average, it requires a single probing period of 147 ms and subsequently 83 ms to process the data. We show the accuracy of this approximation and its effectiveness when applied to the shared-cache management principle of providing isolation so that an appropriate amount of cache space can be allocated. Performance improvements of up to 27% were achieved.

5.1 Introduction

Numerous researchers have proposed using Miss Rate Curves (MRCs) for the purpose of improving management of the memory hierarchy, including file buffer management [Kim et al. 2000; Patterson et al. 1995; Zhou et al. 2001], page management [Azimi et al. 2007; Yang et al. 2006; Zhou et al.

[1]© ACM, 2009. This chapter is a minor revision of the work published in Proceedings of the 14^{th} International Conference on Architectural Support for Programming Languages and Operating Systems 2009 (March 7–11, 2009), http://doi.acm.org/10.1145/1508244.1508259

2004], and L2 cache management [Qureshi and Patt 2006; Stone et al. 1992; Suh et al. 2004]. MRCs capture the miss rate as a function of memory size for a process or a workload consisting of a set of processes, such as a virtual machine, at a particular point in time. MRCs thus identify the memory needs of processes, allowing for intelligent provisioning of these scarce resources.

MRCs can be obtained *offline* in a relatively straightforward way by running the target application or workload multiple times, each time using a different memory size or cache size. While *online* capturing of MRCs for file systems is also relatively easy, say using ghost buffers [Patterson et al. 1995], the *online* capture of MRCs for main memory or for caches is significantly more challenging without hardware support. Nevertheless, Zhou et al. demonstrated how MRCs for main memory can be obtained dynamically in software on commodity hardware [Zhou et al. 2004].

In this chapter, we target L2 caches and introduce **RapidMRC**, a software-based, *online* technique that approximates L2 MRCs on commodity systems with low overhead for the purpose of provisioning shared caches. The challenges in this new context, compared to main memory and disks, are that (1) the potential gains from reducing cache misses are relatively small compared to the gains from reducing main memory misses, and (2) the costs of tracking misses and generating MRCs are large compared to the cost of the miss event. A main memory miss allows plenty of time for the processor to perform tracking and calculations before receiving the data from disk. In contrast, the act of recording a cache miss can be several times more expensive than the cache miss itself.

We accomplish three goals in this chapter. First, we present RapidMRC, a software-based online method to characterize the cache requirements of processes on a commodity processor by generating L2 MRCs in a low-overhead, low-latency manner. We demonstrate how to exploit the available architectural support in modern commodity processors in the form of performance monitoring units (PMUs) to extract information and process it, thus enabling online optimizations at various software levels, such as at the operating system, virtual machine monitor, and programming language run-time system level. We compare the accuracy of online RapidMRC to the real MRCs for 30 applications taken from standard benchmarks.

As the second goal, we examine the multitude of factors in modern processors that can impact the accuracy of the calculated MRC. We examine existing architectural support as well as barriers in developing the RapidMRC technique.

As the third goal, we show how RapidMRC can be applied to provisioning the shared L2 cache by determining the best partition size to allocate to each application running in a co-scheduled manner on a shared cache multicore processor.

5.2 Background and Related Work

In this section, we review miss rate curves, describe the specific requirements for generating L2 cache miss rate curves, and describe related work on L2 cache partitioning.

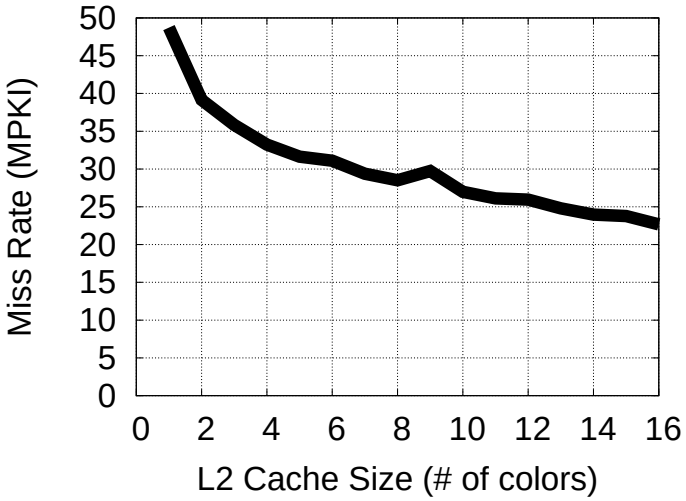


Figure 5.1: The L2 miss rate curve (MRC) of `mcf`, obtained in an offline manner. Generally, as more cache space is given, the miss rate decreases.

5.2.1 Miss Rate Curves

The Miss Rate Curve (MRC) of a memory access sequence identifies the miss rate as a function of the amount of memory allocated to the sequence at a particular point in time. The key advantage of the MRC model over the traditional working-set model is that the MRC model presents an entire trade-off spectrum between allocated memory size and resulting miss rate [Denning 1968]. In contrast, the working-set model only indicates the amount of memory that a process must have for acceptable performance, and it does not identify how performance is affected if the amount of memory allocated is less than its working-set size.

MRCs can be generated for any level in the memory hierarchy. In our study, we focus on generating MRCs for on-chip shared L2 caches. Figure 5.1 shows an example of the L2 MRC of `mcf` obtained in an offline manner, from the SPECcpu2000 benchmark suite, where the L2 cache is divided into 16 colors, using the software-based cache-partitioning mechanism described in Chapter 4[2]. The measured L2 miss rate is plotted as a function of the partition size (number of colors) allocated to the application. The miss rate is measured in terms of the number of misses per thousand completed instructions (MPKI). The general trend in nearly all MRCs is that the miss rate decreases as more space is allocated.

In addition to the specific patterns in memory access sequences, the MRC is affected by the replacement policy of the cache. That is, the MRC of a least-recently-used (LRU) policy may be significantly different from that of a most-recently-used (MRU) policy for the same memory access sequence. Throughout this chapter we assume that the default replacement policy is least-recently-used (LRU) since it is the most commonly used replacement policy for processor caches[3].

[2]In the remainder of this dissertation, the cache-partitioning mechanism includes the additional modifications subsequently described in Section 4.6.4 to eliminate the need for a `MaxTry` threshold.

[3]Due to the high costs of implementing the LRU policy in silicon, processors typically implement a pseudo-LRU

A common method to calculate MRC is the *Stack Algorithm*, originally developed by Mattson et al. [Mattson et al. 1970] and independently by Kim et al. [Kim et al. 1991], both intended for offline analysis of main memory access patterns at page-level granularity. In this algorithm, an *LRU stack* is maintained, consisting of memory addresses generated from the sequence of memory accesses so that the top element is the most recently accessed memory address and the bottom of the stack is the least recently accessed memory address. On each access, the *distance* of the current location of the accessed address from the top of the LRU stack (i.e., *Stack Distance*) is determined before moving the address to the top of the stack. The stack distance of the memory access can be used to speculate whether the access would result in a miss or a hit given a certain memory size. That is, for any memory size larger than the stack distance, the access would be a regarded as a hit since it is expected that the memory element has, at the time of the access, not yet been replaced by the LRU algorithm. On the other hand, for any memory size smaller than the stack distance, the memory access would be regarded as a miss. In order to generate the MRC, a histogram, $Hist$, is calculated where $Hist(dist)$ shows the total number of memory accesses with a stack distance of $dist$. Therefore, the number of misses for memory of size $size$, $Miss(size)$ can be calculated as follows:

$$Miss(size) = \sum_{dist=size+1}^{\infty} Hist(dist) \tag{5.1}$$

Miss rate curves have been used in the past to manage main memory pages [Azimi et al. 2007; Yang et al. 2006; Zhou et al. 2004]. In this context, accesses to pages can be trapped into the kernel and are thus easily seen, and any page replacement policy can be used. Miss rate curves have also been used to manage disk buffer caches [Kim et al. 2000; Patterson et al. 1995; Thiebaut et al. 1992; Zhou et al. 2001] and database application buffer caches [Soundararajan et al. 2008]. In this chapter, we apply miss rate curves to L2 caches by first obtaining L2 cache access traces with the help of hardware PMUs.

In our application of RapidMRC for provisioning the shared cache, we must compare current miss rates across memory access sequences of concurrently executing applications. For this purpose, we normalize the value of $Miss(size)$ over a fixed probing period, using the number of misses per kilo instructions (MPKI):

$$MPKI(size) = 1000 \times \frac{Miss(size)}{CPUInstructions} \tag{5.2}$$

where $CPUInstructions$ is the length of the probing period, measured in terms of the number of instructions executed.

policy, which closely approximates true LRU [Al-Zoubi et al. 2004]. For example, the POWER5 processor implements a pseudo-LRU policy known as pairwise-compare [Zhang et al. 2008].

5.2.2 L2 MRC Generation

A basic requirement for building a precise LRU stack for computing an MRC is having an accurate trace of the application's memory accesses, which can be obtained in several ways. One way to capture memory traces is to run the application in a simulation environment, where the simulator is able to monitor the execution of individual instructions of the application. This method is extensively used in computer systems research for offline analysis of memory access patterns. However, simulation is not suitable for online use because of its high constant overhead.

Another method of capturing memory traces is to *instrument* all memory access instructions of the application so that the accessed addresses are recorded into a *trace log*. Instrumentation tools such as ATOM [Srivastava and Eustace 1994], Pin [Luk et al. 2005], DynInst [Buck and Hollingsworth 2000b], Dynamo [Bala et al. 2000], DynamoRIO [Bruening et al. 2001], and JIFL [Olszewski et al. 2007] can be used for this method. While being simple and straightforward to implement, this approach is too expensive for online use when all memory accesses are instrumented. It substantially slows down the execution of applications (in some cases by a factor of 10 [Seward and Nethercote 2005]) because of the additional instructions that must be executed and poorer instruction cache performance due to the increased instruction footprint.

One way to reduce this overhead is to dynamically enable or disable instrumentation in an on-demand basis using a dynamic code modification system such as DynamoRIO. However, there still exists a fixed overhead with such a system. Zhao et al. have reported an average minimum runtime overhead of 13% for DynamoRIO when instrumentation (for purposes such as memory access tracing) is disabled [Zhao et al. 2007c]. In contrast, our approach has no overhead when memory access tracing is disabled.

Our method for collecting memory access traces uses features available in modern PMUs in a way that requires no changes to applications and has sufficiently low overhead so as to be useful for online purposes. Our software-based solution is in contrast to hardware-based solutions that have been proposed in the past. For example, Qureshi and Patt [Qureshi and Patt 2006] and Suh et al. [Suh et al. 2004] propose hardware additions to future processors to obtain L2 MRCs online. Their general strategy is to monitor several cache sets in an N-way set-associative cache by attaching access counters to each LRU position within a set. Within each set, these access counters serve the role of tracking the stack distance in Mattson's algorithm.

Zhao et al.'s ubiquitous memory introspection framework offers a potential alternative platform for achieving our goal of obtaining MRCs online, however, there are some challenges in their approach [Zhao et al. 2007c]. The first challenge is the previously mentioned constant minimum runtime overhead of 13%. In contrast, our PMU-based approach has no runtime overhead when memory tracing is disabled. The second challenge is that memory tracing would be more expensive than in our PMU-based approach because traces would be longer. This longer length is necessary because every load/store operation is captured rather than only L2 cache accesses, as done in our

PMU-based approach (described in Section 5.3.1). The third challenge is that calculating the MRC is more expensive than in our PMU-based approach because hardware behaviour would need to be simulated in software. For our purposes of obtaining MRCs online, this behaviour would be the L1 cache behaviour because all L2 accesses are first filtered through the L1 cache. That is, in order to determine which accesses reach the L2 cache, we must determine which accesses cause misses in the L1 cache. In contrast, our PMU-based approach can directly capture the L2 cache accesses by appropriately configuring the PMU. In general, the fundamental advantage of a PMU-based approach over a dynamic instrumentation approach is that the PMU can capture just the hardware events that are of interest, which are typically a small fraction of all encountered hardware events, potentially leading to lower costs and simpler designs and implementations.

The work by Lu et al. [Lu et al. 2004] is closer in spirit to our work while trying to achieve the same goals as Zhao et al.. Their work is closer in spirit because of their extensive use of PMUs. They make use of the Intel Itanium 2 PMU to obtain hot traces and perform various compiler-oriented dynamic code optimizations. Due to limitations in the Itanium 2 PMU capabilities, they did not attempt to perform data address tracing as we will do in this chapter. In contrast to their more general, all encompassing framework, we have a very specific purpose of capturing L2 cache accesses for MRC generation.

Several researchers have presented various analytical models to calculate L2 cache miss rates based on memory access traces [Berg and Hagersten 2004; Guo and Solihin 2006; Shen et al. 2007]. These techniques were mainly targeted for offline analysis of memory access traces obtained using a simulator. For Solihin et al.'s model to be used online, additional hardware support would be required [Guo and Solihin 2006]. Shen et al.'s model does not require additional hardware support to be used online but they have yet to show its use in an online environment [Shen et al. 2007]. Berg and Hagersten's model can also be used online without additional hardware support [Berg and Hagersten 2004], and they have subsequently shown how to obtain the reuse distance using watchpoints on commodity processors with an average overhead of 39% throughout the entire execution of an application [Berg and Hagersten 2005]. In contrast, our work takes an approach opposite to Berg and Hagersten's sampling-based approach over the entire execution of the application because we attempt to capture every access for a short window of accesses for online optimization purposes.

Solihin et al. also present a technique to model the miss rates of an L2 cache using *circular sequence profiling*, which is similar to Berg and Hagersten's reuse distance [Guo and Solihin 2006]. The main focus of their work was to model the impact of various cache line replacement policies for a better offline exploration of hardware cache designs. Again, circular sequence information is not obtainable in current processors and hence would require hardware additions to future processors. Since their focus is on offline cache design exploration, their technique was never intended for online use.

5.2.3 L2 Cache Partitioning and Provisioning

In support of the hardware-based partitioning mechanisms described in Section 4.2, architecture researchers have proposed further hardware extensions to capture cache behaviour at runtime and aid in determining *optimal* partition sizes. In addition to the hardware extensions described in Section 4.2, Zhao et al. propose hardware extensions to enable fine-grained monitoring of cache events such as occupancy, interference, and sharing, in order to characterize application cache usage, improve performance, provide quality-of-service guarantees, and provide metering for billing purposes [Zhao et al. 2007a]. Settle et al. propose similar hardware additions but only for detecting and preventing potential interference, by monitoring the number of accesses and misses to groups of cache sets [Settle et al. 2004]. Buck and Hollingsworth propose hardware additions to monitor cache evictions more precisely, capturing the address of the evicted data [Buck and Hollingsworth 2006]. Dybdahl et al. have also investigated additional hardware extensions [Dybdahl et al. 2006 2007], as have Srikantaiah et al. [Srikantaiah et al. 2009b]. Nikas et al. investigated using hardware Bloom filters to more efficiently monitor cache usage of applications for the purposes of provisioning [Nikas et al. 2008]. Earlier explorations include Zilles and Sohi [Zilles and Sohi 2001], and Collins and Tullsen [Collins and Tullsen 2001].

In general, hardware proposals for cache partitioning and monitoring have the inherent advantage of lower runtime overheads and better accuracy than software implementations, but it remains to be seen if and when these proposals will appear in real processors, thus giving software solutions the practical advantage of being deployable today. However, software-based solutions currently provide only cache partitioning and not cache monitoring. Consequently, they are missing an important piece of the puzzle: how to provision the shared cache in an online manner with low overhead. For software-based mechanisms, only trial and error techniques for provisioning have been employed so far, although they typically use a form of binary search to reduce the number of trials [Kim et al. 2004; Lin et al. 2008]. With these approaches, determining the best sizes for more than 2 applications or cores is non-scalable because the number of possible size combinations grows exponentially with the number of applications or cores[4]. Table 5.1 lists the number of combinations for 1 to 16 applications, given a cache with 16 colors, such as in the IBM POWER5 multicore processor. Similarly, Table 5.2 lists the number of combinations given a cache with 64 colors, such as in the Intel Xeon multicore processor [Lin et al. 2008]. With just 5 applications running in a multiprogrammed fashion, there are over a thousand partition size combinations to be tried if this workload was run on an IBM POWER5 multicore processor, and over half a million combinations on an Intel Xeon multicore processor. Using dynamically obtained MRCs, on the other hand, we can eliminate this trial and error approach. A convenient property of MRCs is that they are unaffected by, and

[4]The number of possible size combinations, given n colors and k applications, in our problem domain is known in the field of combinatorial mathematics as the number of compositions of positive integer n into exactly k terms [Weisstein], where $n, k \in \mathbb{N}$, $n \geq k$, and is given by the binomial coefficient $\binom{n-1}{k-1}$.

# of Apps	# of Combos
1	1
2	15
3	105
4	455
5	1,365
6	3,003
7	5,005
8	6,435
9	6,435
10	5,005
11	3,003
12	1,365
13	455
14	105
15	15
16	1

Table 5.1: The number of partition size combinations given a cache with 16 colors, such as in the IBM POWER5 processor.

# of Apps	# of Combos
1	1
2	63
3	1,953
4	39,711
5	595,665
6	7,028,847
7	67,945,521
8	553,270,671
.	.
..	...
...	....
32	916,312,070,471,295,267
.	.
..	..
...	...
64	1

Table 5.2: The number of partition size combinations given a cache with 64 colors, such as in the Intel Xeon processor.

independent of, the currently configured cache partition size, because MRCs are generated from a trace of load/store memory operations, regardless of whether these accesses result in a hit or a miss in the L2 cache. In Section 5.4, we apply RapidMRC to provide a practical analytical approach to determining the optimal cache partition size, capable of running on commodity processors.

5.3 Design and Implementation of RapidMRC

In this section, we describe the design and implementation of RapidMRC. We describe how we collect memory access traces and how we generate MRCs from the collected traces. We also discuss important details about the implementation of RapidMRC on the IBM POWER5 processor.

5.3.1 Collecting Memory Access Traces

Our method for collecting memory access traces is based on using data sampling features available in some of the performance monitoring units (PMUs) of existing processors. The key advantage of this method is that it is completely transparent to the application, requiring no code instrumentation, since the process of recording data addresses is done entirely in hardware. The basic PMU feature required is the capability of recording the data address of memory accesses to a data address register (DAR) or to a designated memory buffer. Systems software can then be notified with an exception when the DAR is updated or when the designated memory buffer overflows. This general method of sampling data addresses was first proposed and subsequently implemented by Buck and Hollingsworth [Buck and Hollingsworth 2004 2000a].

Note that we target only the application data residing in the L2 cache but not application instructions. In our work, we assume that instructions have limited impact on L2 cache miss rates and vice-versa. Our results show that this assumption is reasonable.

We configure the PMU to record data accesses to the L2 cache for a short period of time. In our case study environment, the events that access the L2 cache are (1) L1 instruction and data cache misses, (2) L1 data write-through accesses, and (3) hardware prefetches. Due to limitations in our hardware, we only track L1 data cache misses, which are generally much less frequent than L1 data cache hits. We do not track L1 instruction cache misses, L1 data write-through accesses, or hardware prefetches. Despite these limitations, we show in our results that the accuracy of RapidMRC remains high.

It is important to ensure that the time interval over which memory accesses are traced is long enough to identify the patterns in the *reuse distance* of individual cache lines. The size of the access trace must be several times as large as the number of the L2 cache lines so that the reuse distance of each cache line can be observed several times.

It should be noted that Mattson's stack algorithm requires a memory trace and cannot work with random samples of L2 accesses. Thus, we cannot use a random sampling-based approach, such as the one used by Berg and Hagersten [Berg and Hagersten 2005], described in Section 5.2.2. In addition, they showed that such a sampling-based approach required an average overhead of 39% for the entire execution of the application. Given the significantly larger latencies required to obtain adequate data, compared to our single probing period latency of 147 ms on average, responding to phase changes may be difficult in such an approach.

POWER5-Specific Issues in Gathering Traces

The PMU in the IBM POWER5 processor, a member of the PowerPC family, can perform data sampling *continuously*, where the *Sampled Data Address Register* (SDAR) is continuously updated by the PMU as memory instructions with operands that match a selection criterion arrive in the processor pipeline. Systems software can sample SDAR values by periodically reading its value, which identifies the data address specified by the last memory operation that matched the given selection criterion. With this method, *all* address operands of memory instructions have a fairly equal chance of being captured.

Although other processors, such as the Intel Itanium 2, AMD Opteron, and IBM POWER4, can perform data address sampling, they cannot do so continuously in order to capture a trace. As for Intel IA-32 processors, we have experimented with the Precise Event-Based Sampling (PEBS) mechanism and found that the lack of *data address* information made address collection challenging.

We exploit the current implementation of the PMU in the POWER5, where one can set the selection criterion for updating the SDAR to be a miss in the L1 data cache, thus capturing accesses to the L2 cache. We then use a separate hardware performance counter to count the number of L1 data cache misses, and assign an overflow threshold of *one* so that an interrupt is raised upon

every L1 data cache miss and thus freezing all PMU registers. Raising an exception on each L1 miss is costly since each exception flushes the processor pipeline and switches the execution context from user-space to kernel-space and back. One can envision a hardware PMU, that automatically records the *data* address trace into a small pre-designated buffer, either within the processor core or in main memory, raising an exception only when the buffer overflows so that the cost of overflow exception is amortized over a larger number of data samples. To the best of our knowledge, the PMU in none of today's mainstream processors provide such a feature for *data* samples. The Intel PEBS PMU provides such a main memory buffer mechanism but not for *data* samples. To deal with this lack of hardware support and high overhead, we limit the period of time over which addresses are gathered.

Intricacies of the POWER5 introduce two sources of inaccuracy in our method. The first is the fact that in a superscalar processor, there may be more than one L1 data cache miss-inflicting load-store in flight, due to multiple instruction issue and out-of-order execution. With two neighbouring L1 data cache misses being serviced in parallel by the two load-store units of the processor, it is possible that one of them does not cause the SDAR to be updated. When the first L1 data cache miss raises an exception, the entire pipeline is flushed, which includes the second in-flight memory instruction. However, since the memory access request for the second access has already been sent to the lower levels of the memory hierarchy, when the memory instruction is re-issued after the exception is handled, it may not miss in the L1 data cache anymore, and therefore, the SDAR would not be updated by the second memory access. Fortunately, our results in Section 5.6 show that the collected trace is sufficiently accurate for the purpose of computing MRCs. For problematic applications, we show the impact of disabling multiple instruction issue and out-of-order execution.

The second source of inaccuracy is due to hardware prefetch requests to the L1 data cache because they do not cause the SDAR to be updated with the address of the prefetch target. As a result, a stale SDAR value is recorded into the access trace, leading to trace segments containing consecutive entries all with the same value. We handle this problem by converting these repetitions into a series of ascending cache lines accesses, thus emulating the value that should have been recorded into the SDAR, based on documentation of how the POWER5 prefetcher operates [Sinharoy et al. 2005]. Fortunately, our results Section 5.6 show that the corrected trace is sufficiently accurate for the purpose of computing MRCs. Moreover, we examine the impact of using the IBM POWER5+ processor that allows us to disable prefetching during the monitoring period.

5.3.2 L2 MRC Generation

In order to generate MRCs, we record the L2 accesses due to L1 data cache misses by appending them to an *access trace log* located in main memory. This can be done either (1) through an exception-handler in software to copy the value in the DAR register to the access trace log, or (2) automatically by hardware that is capable of directly recording accesses into a designated main memory buffer that serves as the access trace log. We then feed the access trace into an LRU

stack simulator which builds the LRU stack and generates the MRC using the Mattson stack algorithm [Mattson et al. 1970]. In our targeted use of RapidMRC, since the L2 MRCs are used to size the partitions of the L2, we limit the size of the LRU stack to 15,360 elements, which is the the number of cache lines in our L2 cache. The LRU stack simulator implementation is based on Reza Azimi's design [Azimi et al. 2007], which uses the *range list* optimization proposed by Kim et al. [Kim et al. 1991].

As we will show in Section 5.6, cache prefetching and missed events on address trace collection have the effect of causing the calculated MRC to be vertically offset from the real MRC. To adjust for this, we vertically shift (transpose) the calculated MRC so that it matches at least one point of the real MRC. Since any point can be used, in practice, this point can be the currently configured cache partition size, since its miss rate can be easily obtained from the processor PMU. This adjustment gives us a generated miss rate curve that is correctly calibrated along the y-axis.

MRCs predict the miss rate for a fully associative cache. While today's L2 caches are not fully associative, they usually have high associativity (e.g., 16-way). This configuration causes the behaviour of the cache to be similar to that of a fully associative cache. We have found this approximation to be adequate for computing MRCs, as we will show in Section 5.6.

Many applications go through several *phases* in their execution. In each phase, the performance of an application, characterized in terms of key performance metrics such as instructions-per-cycle (IPC), is fairly stable. However, the performance characteristics of two phases of a single application may be substantially different. As a result, we need to take into account the potential changes in an application's MRC caused by phase transitions. While the number of unique phases in an application is often quite small, there may be many transitions back and forth between these phases.

Figure 5.2 shows the impact of phase transitions on the measured MRC of `mcf` as an example. These measurements were taken by using our software-based cache partitioning mechanism described in Chapter 4 on our POWER5 system (Table 5.3), running the application 16 times, each time with a different L2 cache size, and using the PMUs to measure the cache miss rate. The measured, time-varying L2 cache miss rates for each possible L2 cache partition size is plotted in Figure 5.2a. For example, the curve labelled `size 1` corresponds to a partition size of $\frac{1}{16}$ of the total L2 cache size, while the curve labelled `size 16` corresponds a partition size of $\frac{16}{16}$. The x-axis shows the execution progress of `mcf` in terms of the number of instructions completed. As the application executes, it can be viewed as moving further towards the right side of the graph. The y-axis indicates the L2 cache miss rate in terms of the number of misses per thousand completed instructions (MPKI). Thus, the graph shows how the L2 cache miss rate varies during application execution. `mcf` oscillates between two phases repeatedly, a phase with relatively high L2 cache miss rates and a phase with relatively low L2 cache miss rates.

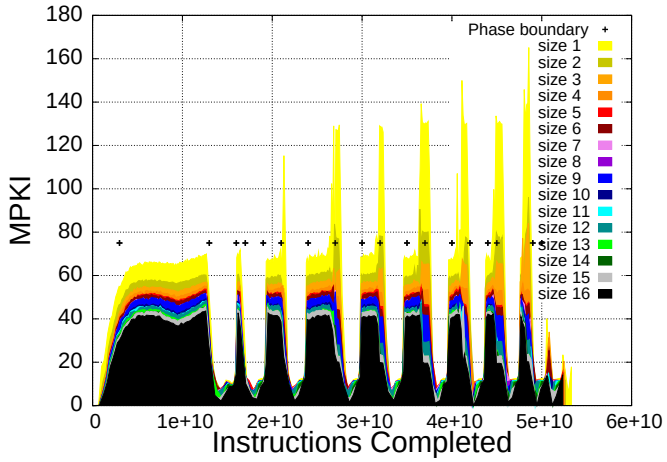
This graph also indicates how the L2 cache miss rate diminishes as the size of the L2 cache partitions is increased. For example, the time-varying miss rate of the `size 16` configuration is

always lower than the time-varying miss rate of the `size 15` configuration, which is always lower than the time-varying miss rate of the `size 14` configuration, etc.. The phase boundary markings in the graph will be described in Section 5.6.2.

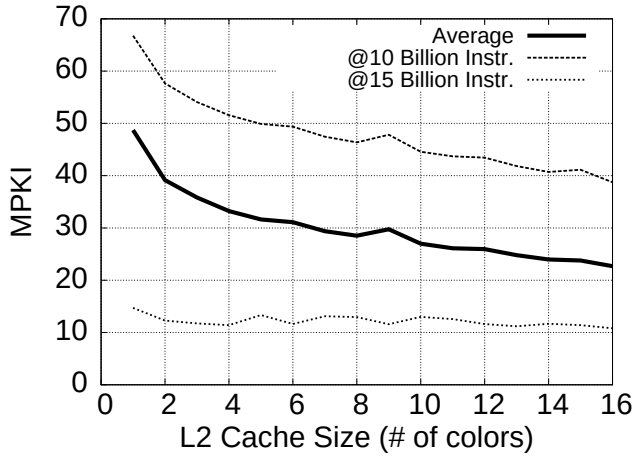
The graph in Figure 5.2b shows the measured MRCs for the two phases of `mcf` compared to the average MRC over the entire execution of the application. The MRCs for the two phases imply substantially different L2 cache requirements within a single application.

Similar to Figure 5.2a for `mcf`, Figure 5.3 to Figure 5.7 show the measured, time-varying nature of the L2 miss rate for other applications across the spectrum of possible cache sizes. Each application was run to completion 16 times, each time with a different L2 cache size.

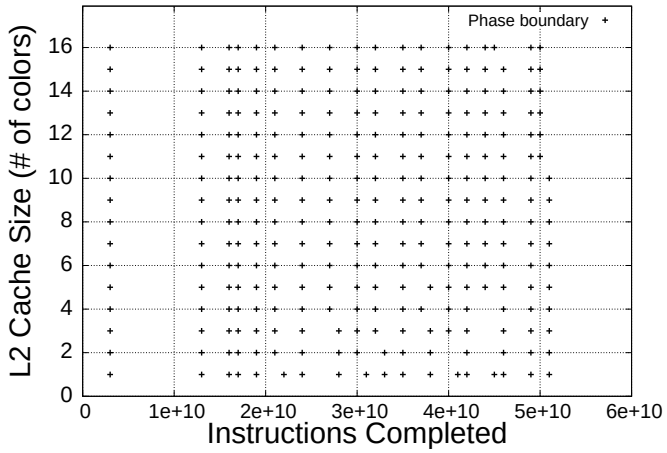
Due to the rapidness of RapidMRC, as will be shown in Section 5.6.2, we have the ability to capture the L2 MRC within phases, as will be shown in our analysis of phase length in Section 5.6.2.



a. The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.

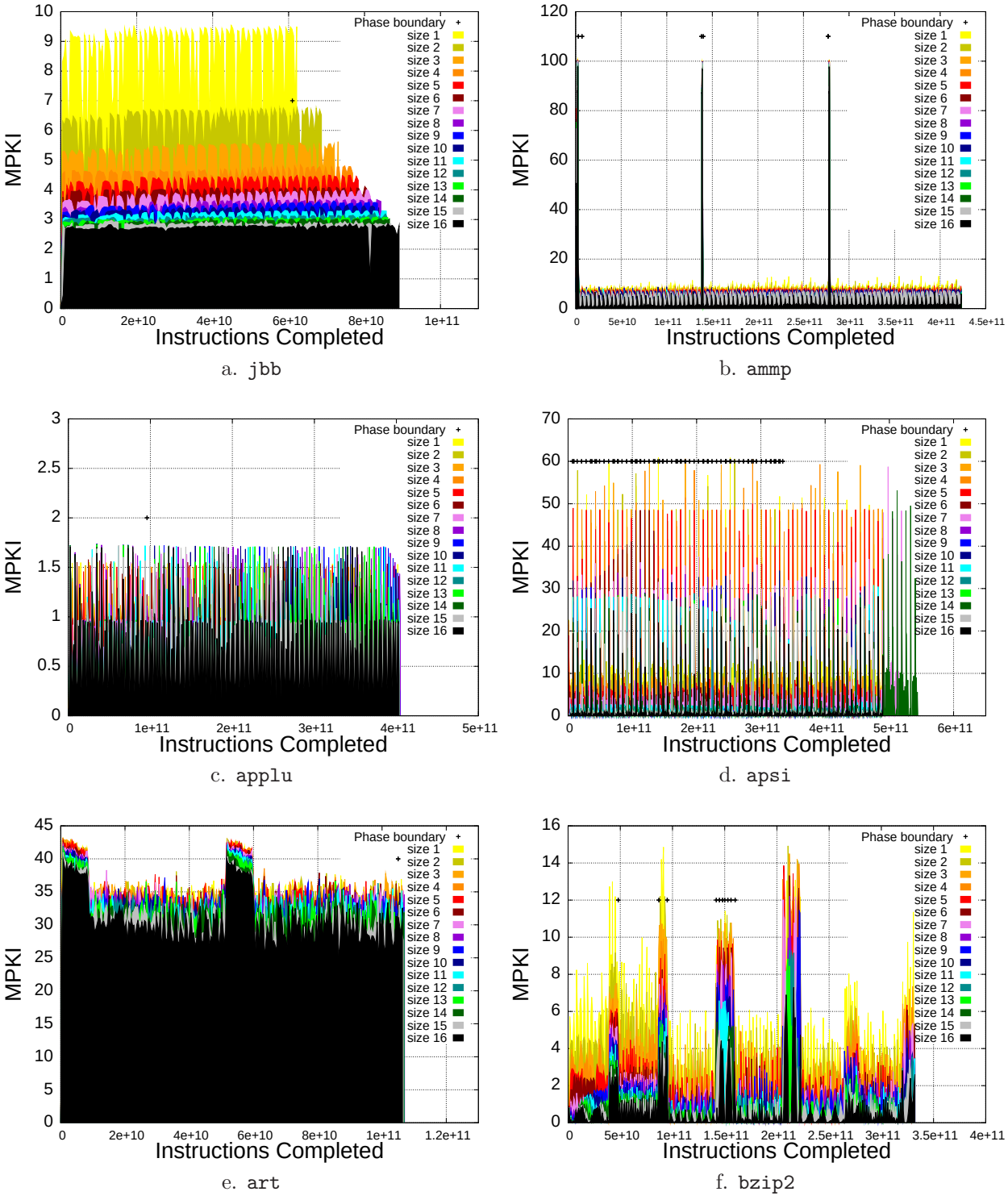


b. L2 MRCs at various execution points.



c. Detected phase boundaries of `mcf` (see Section 5.6.2).

Figure 5.2: Phase transitions in `mcf` and their impact on the L2 MRC, measured on an IBM POWER5 in an offline manner. (a) illustrates the various phases over the execution of the program. (b) illustrates that the MRCs of different phases vary considerably and imply substantially different L2 cache requirements. (c) illustrates that phase boundaries can be detected independently of the currently configured L2 cache partition size since all sizes lead to roughly the same detected boundary locations.

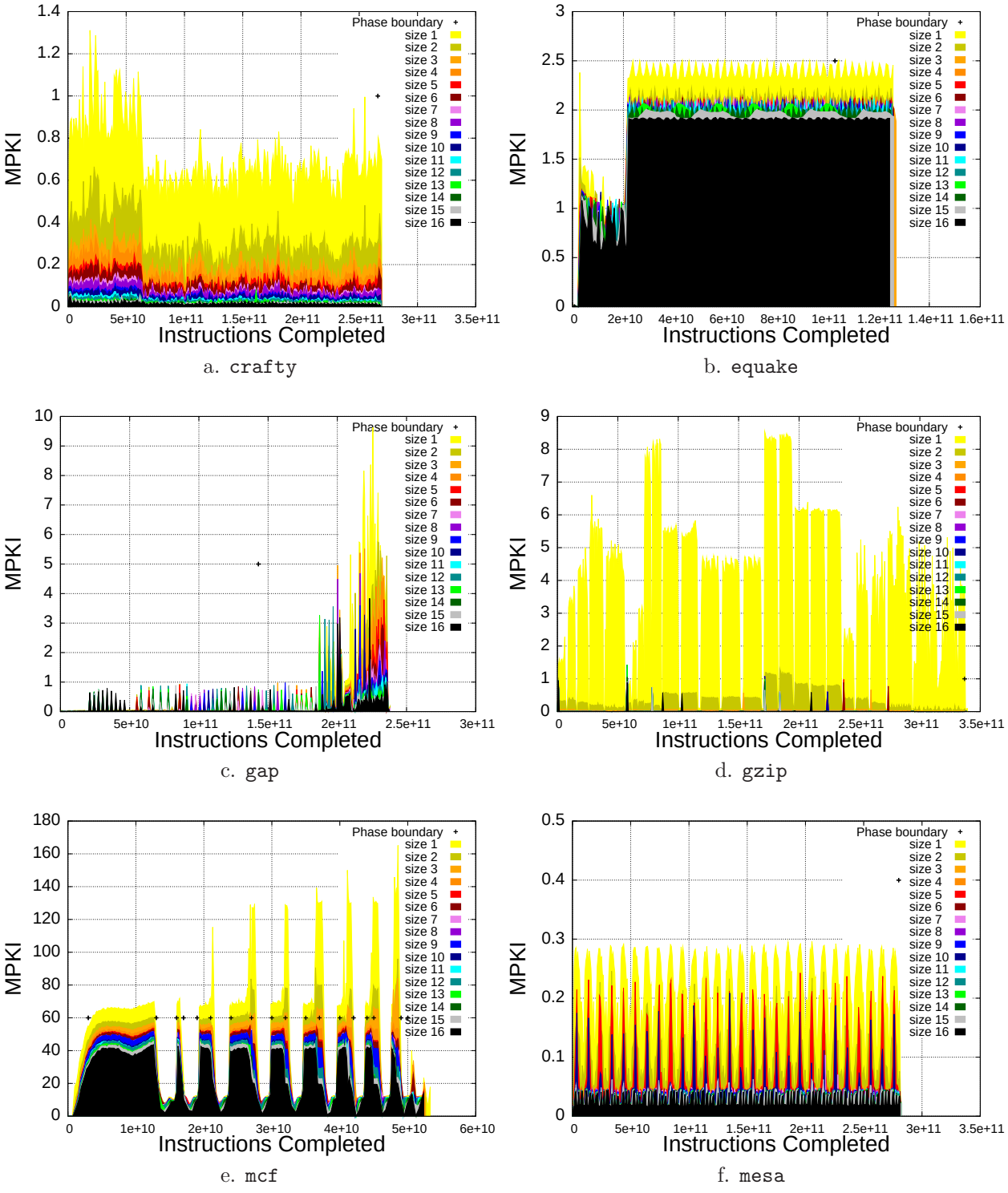


a. `jbb` b. `ammp`

c. `applu` d. `apsi`

e. `art` f. `bzip2`

Figure 5.3: The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes, obtained in an offline manner. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.



a. `crafty` b. `equake`

c. `gap` d. `gzip`

e. `mcf` f. `mesa`

Figure 5.4: The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes, obtained in an offline manner. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.

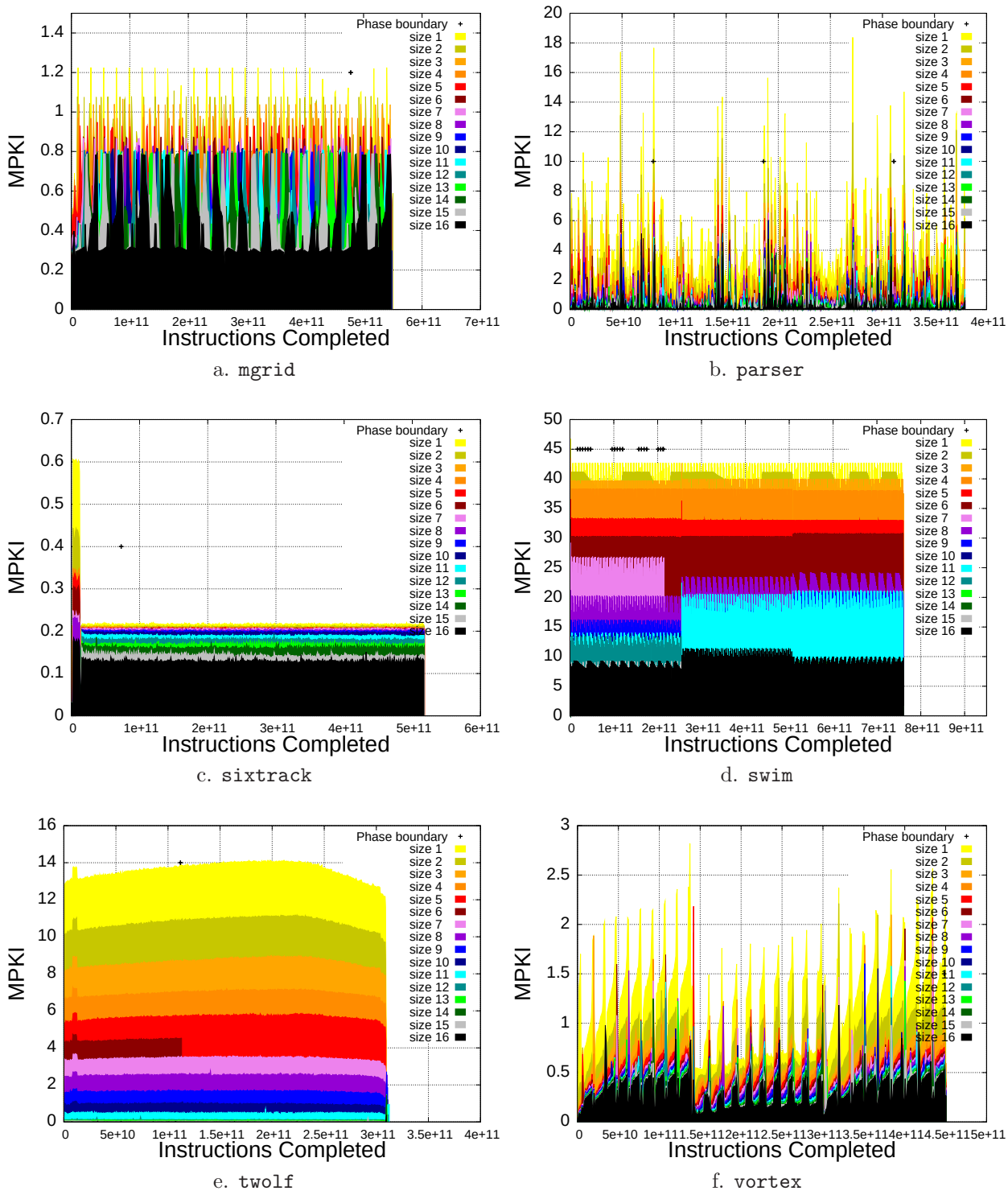
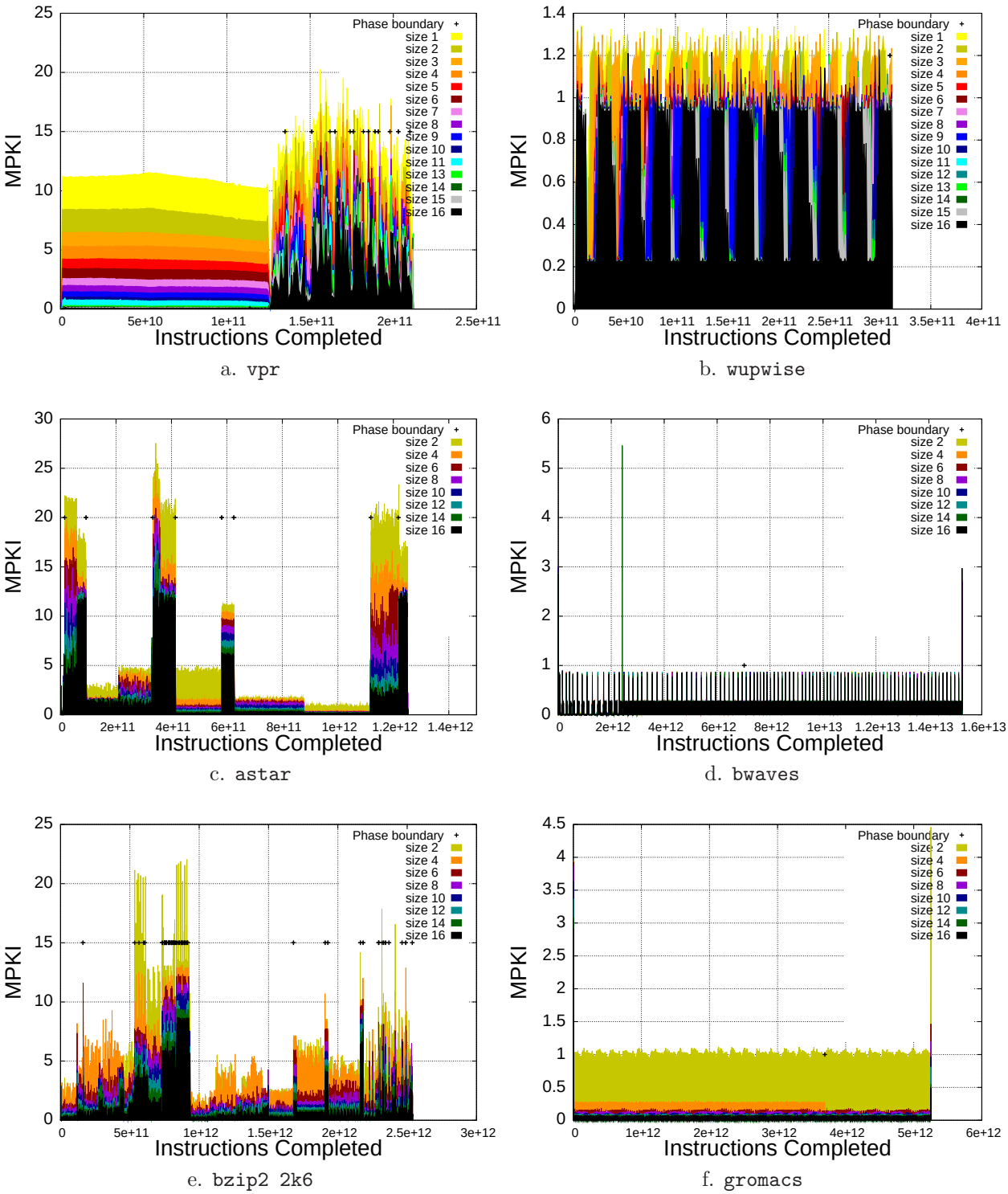


Figure 5.5: The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes, obtained in an offline manner. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.



a. `vpr`

b. `wupwise`

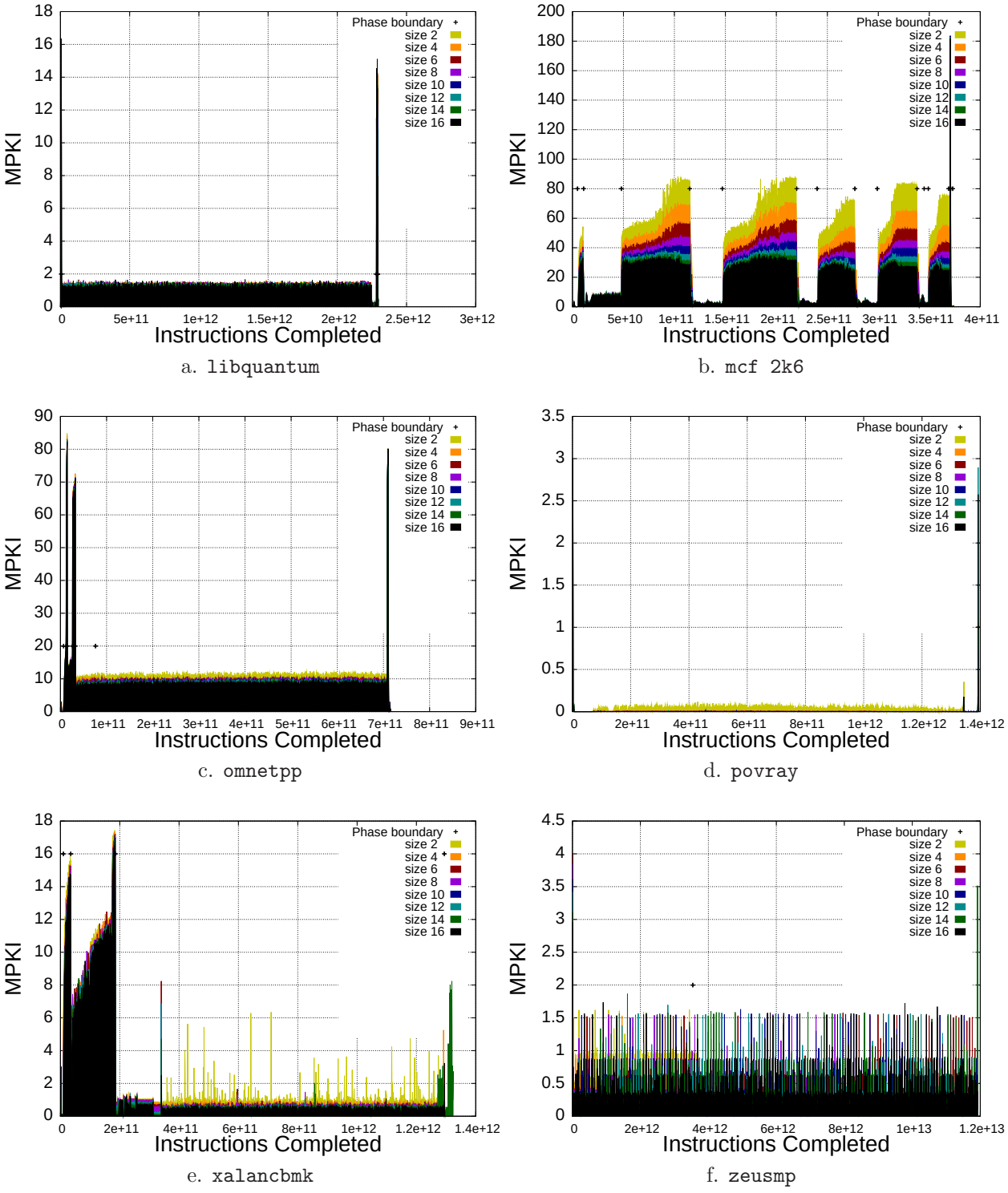
c. `astar`

d. `bwaves`

e. `bzip2 2k6`

f. `gromacs`

Figure 5.6: The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes, obtained in an offline manner. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.



a. `libquantum` b. `mcf 2k6`

c. `omnetpp` d. `povray`

e. `xalancbmk` f. `zeusmp`

Figure 5.7: The measured L2 cache miss rate as a function of time (in instructions completed), for various L2 cache sizes, obtained in an offline manner. Curves labelled `size` N correspond to a cache partition size of $\frac{N}{16}$ of the total L2 cache size.

5.4 Using RapidMRC to Provision the Shared Cache

In this section we describe how RapidMRC can be applied to the problem of provisioning a shared cache, especially in the context of multicore chips. We utilize our software-based cache partitioning mechanism, described in Chapter 4, to divide the L2 cache into a number of *colors*. Each application is allocated a number of colors and as a result can only populate a fraction of the cache. A key issue is how to determine the number of colors to allocate to each application.

For deciding optimal cache provisioning between two co-scheduled applications, we use a simple function which minimizes overall misses in the system. For two processes, a and b, given their miss rate curves MRC_a and MRC_b, either from RapidMRC or from exhaustive offline acquisition, we apply the following size selection function, which returns the value of x:

$$SizeSelection(a,b) = \min_{x \in [1, C-1]} \Big[MRC_a(x) + MRC_b(C-x) \Big] \tag{5.3}$$

where C is the total number of colors into which the cache can be divided. While, for typical C values (e.g., 16), this utility function is sufficiently lightweight to be re-computed dynamically (online) for different phases of applications, in our prototype implementation we compute this utility function for any pair of applications statically (offline).

Our simple method for obtaining the optimal partitioning is effective only for two applications running simultaneously. In configurations where there are more than two applications (more than two MRCs), one can use more sophisticated methods such as the approximation presented by Qureshi et al. [Qureshi and Patt 2006] to address the NP-Hard complexity of the problem [Rajkumar et al. 1997].

Also note that we have defined *SizeSelection* so as to maximize overall performance, however, the operating system could pursue other performance objectives, such as providing quality-of-service, service-level agreements, and process-level priorities.

5.5 Experimental Setup

The experimental results we present here were obtained on an IBM POWER5 system, as specified in Table 5.3, and on a similarly configured POWER5+ system for some experiments. Each POWER5 chip contains an L2 cache that is shared between 2 cores. Each core contains a private L1 data cache and L1 instruction cache. Connected to each chip is an off-chip L3 victim cache, which is also shared between the 2 cores. For some results, such as when disabling the L1 data cache hardware prefetcher, an IBM POWER5+ system was used. It has a similar configuration as the IBM POWER5 system except for having 4 GB of RAM.

RapidMRC was implemented in the Linux Operating System, kernel version 2.6.15 on the POWER5 system. On the POWER5+ system, we used Linux kernel version 2.6.24 running in Bare-Metal Linux mode without the embedded POWER hypervisor [Venton et al. 2005]. Our

Item	Specification
# of Cores per Chip	2
Frequency	1.5 GHz
L1 ICache	64 kB, 128-byte lines, 2-way set-associative, per core
L1 DCache	32 kB, 128-byte lines, 4-way set-associative, per core
L2 Cache	1.875 MB, 128-byte lines, 10-way set-associative, per chip
L3 Victim Cache	36 MB, 256-byte lines, 12-way set-associative, per chip, off-chip
RAM	8 GB (4 GB on POWER5+)

Table 5.3: IBM POWER5 specifications.

Parameter	Value	Referenced Section
LRU stack size	15,360 elements	5.3.2
Trace log size	163,840 entries	5.6.3
Stack warmup period	Auto-detect or 81,920 entries	5.6.4
Application window	Begin after 10 billion completed instructions	5.6.1

Table 5.4: LRU stack simulator specifications.

modifications to Linux consist of approximately 150 lines of code (LOC), stemming from 25 LOC for collecting the memory access trace, 100 LOC for generating the L2 MRC using Mattson's stack algorithm, and 25 LOC for implementing the *SizeSelection* function.

RapidMRC was evaluated using 19 applications from SPECcpu2000, 10 applications from SPECcpu2006, and SPECjbb2000. For SPECcpu2000, we were unable to successfully compile the remaining 6 applications, which were mostly Fortran-based. For SPECcpu2006, the 10 applications (out of 29 applications) were chosen solely based on which applications had already been previously compiled into 64-bit POWERPC Linux binary format by our research group. Although we did not intentionally leave out applications from the benchmark suites to bias the results in our favour, we recognize that there is the possibility of unintentional biasing. The IBM J2SE 5.0 JVM was used to run SPECjbb2000 with 1 warehouse configured. For SPECcpu2000 and SPECcpu2006, the applications were run using the standard *reference* input set. Thread migration between cores was disabled in the operating system to provide a more controlled execution environment.

The values of the LRU stack simulator parameters are shown in Table 5.4. These values were set intelligently, as described in the indicated sections, and parameter sensitivity is explored in subsequent sections.

5.6 Results

We begin by evaluating the accuracy of RapidMRC by comparing it to the real MRC values, and then analyzing the run-time overhead. We then look at the impact of various factors on RapidMRC. Finally, we briefly present results from applying RapidMRC to sizing cache partitions.

Column	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
Workload	Trace Log. Time ($x10^6$ cycs)	MRC Calc. Time ($x10^6$ cycs)	App. Instrs. ($x10^6$)	Avg. Phase Length instrs : cycs ($x10^9$)	Pre-fetch Conver-sion (% Log)	% Log Used for Warm-up	LRU Stack Hit Rate	Vert. Shift (MP-KI)	Distance (MPKI) 160k Log	Distance (MPKI) 1600k Log
jbb	189	86	17	60 : 101	15 %	42 %	80 %	1.3	0.51	0.51
ammp	192	72	22	46 : 65	14 %	83 %	95 %	1.6	1.02	1.02
applu	201	83	27	400 : 483	7 %	29 %	70 %	-1.6	0.28	0.26
apsi	462	59	351	5 : 6	39 %	60 %	88 %	1.1	1.09	1.09
art	177	146	6	100 : 246	18 %	20 %	76 %	17.5	4.54	4.06
bzip2	200	81	26	16 : 17	4 %	81 %	97 %	-0.8	1.02	0.94
crafty	191	48	24	250 : 249	5 %	50 %	98 %	0.0	0.08	0.07
equake	252	128	57	120 : 150	42 %	12 %	48 %	-0.6	0.12	0.12
gap	599	98	301	175 : 224	76 %	27 %	65 %	-0.2	0.00	0.00
gzip	191	51	21	325 : 446	30 %	50 %	99 %	-0.1	0.14	0.22
mcf	185	155	5	3 : 11	2 %	13 %	50 %	25.0	2.57	2.64
mesa	284	47	91	275 : 356	7 %	50 %	98 %	0.0	0.03	0.03
mgrid	192	69	30	550 : 509	54 %	38 %	72 %	-1.2	0.08	0.07
parser	203	59	24	104 : 144	5 %	50 %	98 %	0.3	0.28	0.21
sixtrack	207	48	36	500 : 474	8 %	50 %	99 %	0.2	0.13	0.12
swim	204	113	20	11 : 28	62 %	15 %	51 %	2.1	6.12	4.88
twolf	191	77	16	300 : 518	4 %	50 %	100 %	2.2	1.72	1.71
vortex	251	74	97	450 : 400	11 %	54 %	88 %	0.0	0.02	0.03
vpr	189	69	16	16 : 25	5 %	50 %	99 %	1.7	1.03	1.01
wupwise	291	137	129	310 : 314	48 %	15 %	37 %	0.1	0.01	0.01
astar	185	158	16	152 : 355	3 %	30 %	69 %	-0.3	0.20	0.19
bwaves	150	62	14	15,000 : 16,088	0 %	50 %	91 %	-0.8	0.00	0.00
bzip2 2k6	161	81	27	42 : 38	11 %	50 %	92 %	0.4	0.43	0.47
gromacs	243	90	71	5,000 : 7,230	11 %	62 %	89 %	-0.2	0.06	0.02
libquantum	153	404	11	2,250 : 1,753	96 %	9 %	0 %	-14.0	0.02	0.02
mcf 2k6	161	282	6	25 : 104	2 %	20 %	53 %	30.1	1.95	1.96
omnetpp	167	323	7	650 : 1,704	0 %	24 %	86 %	-15.8	6.57	3.82
povray	161	324	24	14,000 : 14,362	6 %	50 %	100 %	0.0	0.00	0.00
xalancbmk	176	177	21	324 : 551	4 %	66 %	88 %	2.1	0.53	0.53
zeusmp	224	113	102	12,000 : 12,650	5 %	47 %	83 %	0.1	0.13	0.15
Average	221	124	54	1,782 : 1,987	20 %	42 %	79 %	3.9	1.02	0.87

Table 5.5: RapidMRC statistics.

5.6.1 MRC Accuracy

There are two components to MRC accuracy: curve shape and vertical offset (v-offset). Matching the shape is the challenging component, whereas matching the v-offset is relatively easy, as described in Section 5.3.2. Factors influencing the v-offset will be examined in later subsections.

The size of the access trace log was configured to 160k entries. For each application, the percentage of the trace log used for warming up the LRU stack is shown in Table 5.5, column f. The number of entries used for warmup was either determined automatically by the MRC calculation engine or it was statically set to 80k entries, which is one half of the trace log length. For automatic warmup determination, we waited until all entries in the LRU stack were occupied before switching out of warm up mode. For some applications, the trace log was not long enough to warm up the LRU stack under this criteria. These applications had very small working set sizes and seldom spilled to memory, as evidenced by the LRU Stack Hit Rate shown in Table 5.5, column g. Therefore, the statically set warmup length was, in fact, adequate for them.

Figure 5.8 illustrates the online calculated MRC compared to the real MRC for each of the

30 applications that we ran. To obtain the real MRCs, we used the exhaustive offline method combined with our software-based cache partitioning mechanism described in Chapter 4. For each of the possible 16 cache sizes of our L2 cache, the application was executed in its entirety while using the processor PMU to measure the L2 cache miss rate every 1 billion processor cycles. The machine was rebooted before each run, leading to the same initial system state before each run. Both real and calculated MRCs are taken from a brief slice of execution, at the 10 billion completed instruction mark. For the real MRCs, the length of the slice is 1 billion completed instructions, whereas the slice length of the calculated MRCs varies and is shown in Table 5.5, column c, averaging to 54 million instructions. Variation occurs because of the varying amount of processor work required by each application to fill the 160k trace log. To verify that the offline real MRC generated from the 1 billion instruction slice of a phase was indeed representative of the entire phase, we also experimented with larger, 10 billion instruction slices and obtained the same results.

For each calculated MRC, v-offset matching was done, as described in Section 5.3.2, using the 8-color point of the real MRC. This shift amount was uniformly applied to all other points of the calculated MRC, resulting in a uniform vertical shift without any distortion to its shape. Table 5.5, column h shows the amount of vertical shifting applied to each application.[5]

Most of the real MRCs in Figure 5.8 show monotonically decreasing cache miss rates as the cache size is increased, as expected. However, there are some exceptions, such as `ammp`, `applu`, `apsi`, and `bzip2`. Interestingly, `ammp` shows anomalies at 5, 10, and 15 colors, which are multiples of 5. As described in Section 4.6.1, we believe that these anomalies are due to two factors. First, each possible cache partition size configuration leads to different virtual-to-physical page mappings, which may cause pressure on various cache sets of the set-associative L2 cache. Second, since cache partitioning was applied only to applications and not the operating system itself, interference from the operating system may result. For example, there may be the presence or absence of cache interference between the application and operating system meta-data, such as the virtual-to-physical page table entries. Such interference may potentially lead to slower resolution of TLB (translation look-aside buffer) misses when the page table entries are not present in the L2 cache and must be obtained from main memory, as illustrated by Soares et al. [Soares et al. 2008]. Finally, these anomalies cannot be due to altered cache miss patterns stemming from physical memory reclamation performed by the operating system, since the machine was rebooted before each run to create the same initial system system before each run.

For 25 out of the 30 applications, the calculated MRCs match closely to the real MRCs. The general trend is that RapidMRC is capable of tracking a variety of shapes from real MRCs. However, there are five problematic applications: `swim`, `art`, `apsi`, `omnetpp`, and `ammp`. Using a longer, 1600k-entry trace log improved `swim`, as shown in Figure 5.9a, but it remains problematic. Some improvements to `art` were achieved, as shown in Figure 5.9b, on the POWER5+ processor config-

[5]The average is calculated using absolute values.

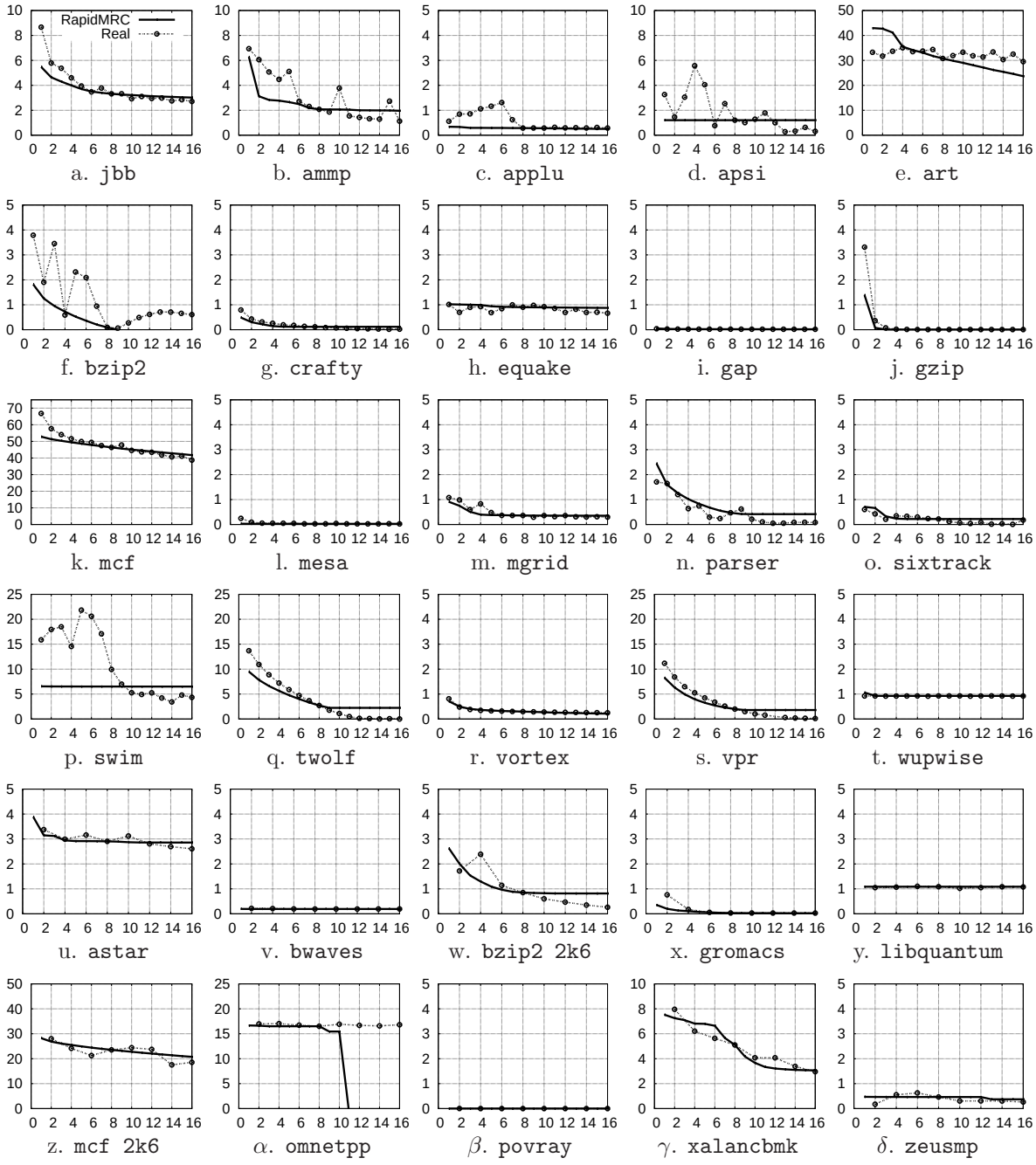


Figure 5.8: Online RapidMRC vs offline real MRCs. x-axis = allocated L2 cache partition size (# of colors), y-axis = resulting L2 cache miss rate (MPKI). For most of the applications, the online calculated MRCs match closely to the real MRCs obtained in an offline manner.

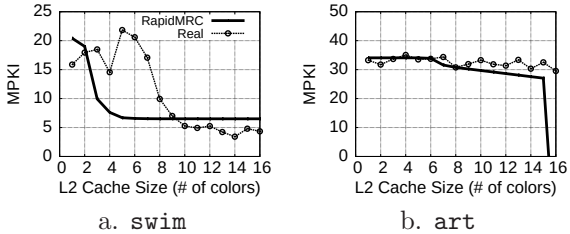


Figure 5.9: Improved RapidMRC.

ured with hardware data prefetching disabled, out-of-order instruction execution disabled so that instructions are executed in-order, and multiple instruction issue disabled so that instructions are issued one-at-a-time. However, a problem remains with the 15-color point. In general, the sources of inaccuracy in these five applications are subject to further research, since they are not caused by the factors examined in the subsequent subsections.

For a quantitative evaluation of MRC similarity, we propose using the metric of average MPKI distance between each real and corresponding calculated point, over the 16 possible cache sizes. The formula is shown below, and the calculated values are shown in Table 5.5, column i.

$$Distance = \frac{1}{16}\sum_{i=1}^{16}|MPKI_{real}(i) - MPKI_{calc}(i)| \tag{5.4}$$

5.6.2 Overheads

Table 5.5, columns a and b show the overheads involved in calculating the MRC. The trace logging time measures the wall clock time required to capture 160k entries into the trace log during application execution. On average, it takes 221 million cycles to obtain the trace log, which is 147 ms on our POWER5 system. During this trace log period, the application is still making progress, although much slower, at 24% of the original IPC on average.

A 160k-entry trace log, with entries of 64-bits each, consumes 1.25 MB of space. Fortunately, the log itself is accessed in a streaming pattern, meaning that (1) segments of the log will be prefetched by hardware to enable fast appends, and (2) these segments will be evicted from the cache in minimal time because they are accessed only once during the tracing period. To restrict its occupation and interference in the L2 cache, cache partitioning could have been applied to restrict the trace log to occupy only a small portion of the cache, although we have not done so. On the POWER5 processor, this portion could be a 120 kB partition of the cache.

We also measured the total interrupt service routine (ISR) time, which is a subcomponent of the trace logging time measurement. The total ISR time is the total wall clock time spent solely in the interrupt service routine to record an L1 data cache miss event into the trace log. Nearly all applications spent a total of 69 million cycles (46 ms) in the interrupt service routine. This value is constant across all applications because there is little variability in the straight-forward task of recording the data address register value into a trace log entry memory location.

The MRC calculation time is the time required to process the trace log and generate the calculated curve. This time was acquired assuming that the application is not running during the calculation. The average time required is 124 million cycles (83 ms). Given the two columns of trace logging time and MRC calculation time, we can see that the average time required to perform online MRC calculation is 345 million cycles (230 ms).

Overheads in Relation to Phase Length

The actual runtime overhead incurred by RapidMRC depends on the frequency of phase transitions, which require re-computation of the MRC. Due to limitations in our prototype implementation of RapidMRC, we did not automatically track program phase transitions and re-trigger RapidMRC. However, we have done post-mortem analysis on the collected PMU data to calculate the average length of application phases. Column d in Table 5.5 indicates the average phase length of each application, both in terms of the number of instructions and the number of processor cycles. In all but two cases (`apsi` and `mcf`) the total runtime overhead of trace logging and online MRC calculation would be below 2%. In many cases, due to very long phases, the overhead would be negligible.

In our post-mortem analysis, to locate phase transitions in our collected PMU data, we used the following simple heuristic. We used changes in the L2 cache miss rate as the indicator of phase transitions because it directly reflects the changes in the application's cache usage, rather than IPC as suggested by Sherwood et al. [Sherwood et al. 2003]. The L2 cache miss rate of an application can be monitored online with negligible overhead. In order to identify *significant* changes in the miss rate, we used the following simple heuristic in our post-mortem analysis. We divided the collected PMU data into intervals containing a fixed number of instructions. At the end of each interval, we compared the miss rate of the current interval against the average miss rate of the past w intervals, and a phase transition was declared if the two miss rates differed more than a specified threshold. In addition, since phase transitions can span several intervals, another threshold is used as the minimum/maximum miss rate difference threshold to signify the beginning/end of a lengthy phase transition.

The numbers shown in Table 5.5, column d were obtained using the following parameter values for the above heuristic: (1) the L2 miss rate of the 8-color cache size configuration, (2) an interval length of 1 billion instructions, (3) a history size of $w = 3$, (4) a miss rate difference threshold of 3 MPKI, and (5) a start/end of phase transition threshold of 50%.

An indication of the accuracy of this heuristic can be seen in the phase boundary markings shown for `mcf` in Figure 5.2a. These boundary locations coincide with the actual phase transitions visually depicted in the graph. Phase boundary markings are also shown for other applications, in Figure 5.3 to Figure 5.7. For applications where only 1 phase boundary was detected, located somewhere in the middle of the timeline (rather than at the right-most end), the detected boundaries were typically due to an unresolved, intermittent problem during the dumping of the PMU buffers to the Linux Syslog facility. This problem would intermittently occur after a measured run had successfully completed, where upon the PMU buffers would then be transferred to the `/var/log/messages` file. For example, in Figure 5.5e, `twolf` configured with a partition size of 6 colors ran successfully to completion, however, upon dumping the PMU buffer to the Linux Syslog facility, only $\frac{1}{3}$ of the PMU buffer contents were transferred before the system crashed. Therefore, only approximately

1×10^{11} instructions worth of PMU data were retained. Analogous problems are seen in `swim`, Figure 5.5d, where the 7 color partition size configuration experienced problems while dumping the PMU buffer to the Linux Syslog facility (approximately 2×10^{11} instructions). Although we used the L2 cache miss rate of the 8-color cache size configuration to determine phase boundaries of an application, our data processing script conservatively processed data only up to the shortest execution length seen in all runs (1 color to 16 colors) of the application.

Figure 5.2c provides an example to demonstrate that these boundary locations are insensitive to an application's currently configured L2 cache size. The graph indicates the phase boundaries of `mcf` detected by monitoring the L2 MPKI for each possible L2 cache size. The x-axis indicates execution time in terms of instructions completed, while the y-axis indicates the configured L2 cache partition size. Along a given y-axis height, the "+" symbols indicate the location of detected phase boundaries. This graph shows that the vast majority of phase transitions are detected at the same points of execution for all L2 cache sizes.

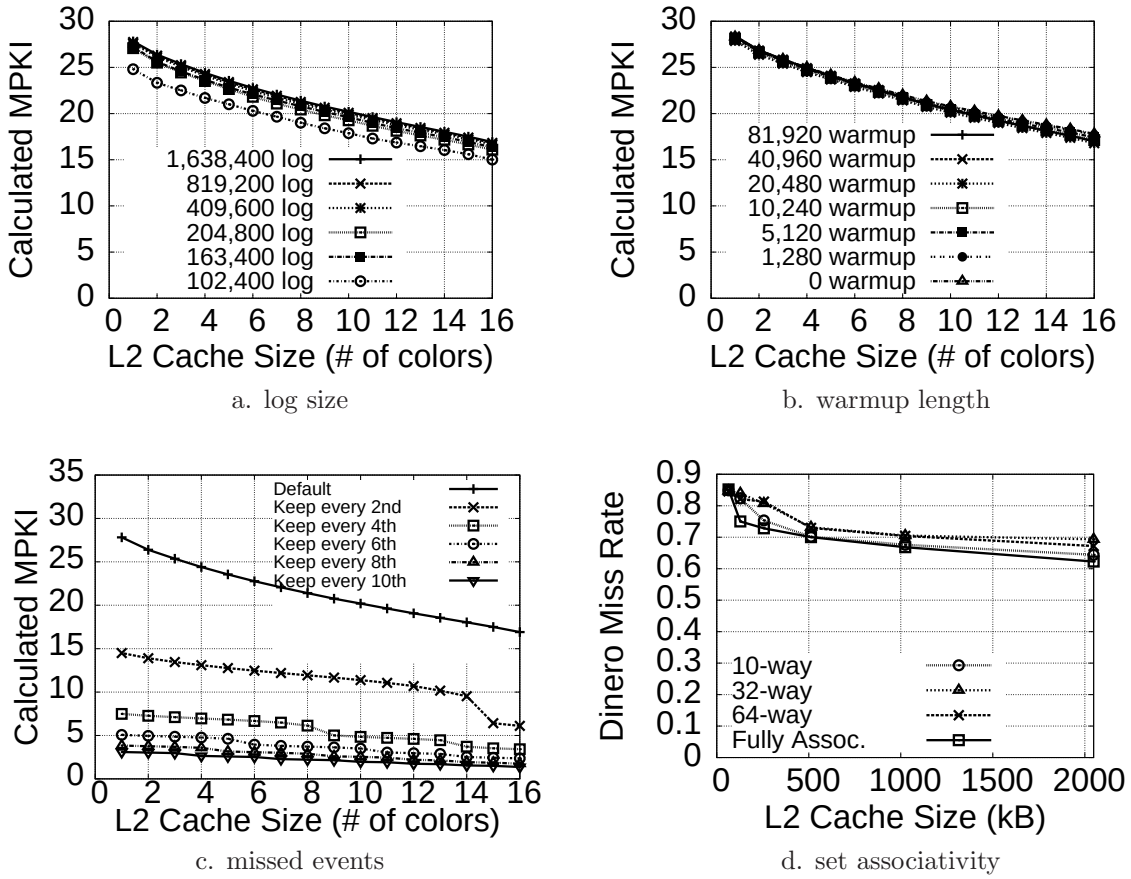
Figure 5.2c also shows that changes to the MRC as a whole, can be detected by monitoring changes to just a single point of the MRC. If a single point on the MRC changes significantly, then all points of the MRC change significantly too. Conversely, if a single point on the MRC does not change, then all points do not change significantly either.

5.6.3 Impact of Trace Log Size

We chose a trace log that was long enough so that the bottom stack position, the furthest from the top of the LRU stack, had a chance of being incremented several times. Since our LRU stack in our experiments was 15,360 in length, in the worst-case cache-hit scenario, it would require a trace log of at least $15,360 + 1$ in length in order for the first trace log entry to end up at the bottom of the stack and then be accessed on the 15,361st access, registering a stack hit. To be conservative, we chose a trace log of approximately 10 times the length of the LRU stack, resulting in our trace log length of 160k entries.

Although the trace log itself can pollute the L2 cache, this impact has been automatically incorporated into the RapidMRC curves of Figure 5.8 and is shown to have little impact on accuracy.

In addition to the 160k-entry trace log size, we also tried using a 1600k-entry trace log size. Although `swim` benefited greatly, as shown in Figure 5.9a, the other applications did not show benefits. Figure 5.10a shows how `mcf` is largely unaffected by the log size. The warmup period is 50% of the trace log size. In order to see the impact on the vertical shifting of the curves, v-offset matching was not applied. For the remaining applications, we show the average MPKI distances for a 1600k-entry log in Table 5.5, column j.



a. log size

b. warmup length

c. missed events

d. set associativity

Figure 5.10: Impact of various factors on the calculated MRC of `mcf`. (a) Our chosen log size of 163,400 entries is adequate. (b) Our chosen warmup length is adequate. (c) Missed events cause the curve to vertically shift downwards and can cause shape distortion. (d) The fully associative cache model does not materially impact the calculated cache miss rates.

5.6.4 Impact of Warmup Period

As with any structure that contains state information, the LRU stack requires a warmup period before it begins recording statistics. This warmup period prevents the stack distance counters from initially reporting incorrect stack position hits, as well as false cold misses. For automatic warmup determination, we waited until all entries in the LRU stack were occupied before switching out of warm up mode. The impact of varying the warmup period for `mcf` is shown on Figure 5.10b. In order to see the impact on the vertical shifting of the curves, v-offset matching was not applied. Similar trends were seen for the other applications and are not shown. From these results, we can see that our chosen criteria for warmup is adequate for MRC accuracy.

5.6.5 Impact of Missed Events

The POWER5 PMU does not guarantee that it will capture every single L1 data cache miss event, as described in Section 5.3.1. In this section, we examine the impact on the calculated MRC of losing more and more of these events. Since we are unable to obtain the number of lost events

from the PMU for these applications, we examine the impact on the calculated MRC by artificially dropping more and more entries from the trace log. By working forward to capture the trend, we can extrapolate the trend backwards.

Figure 5.10c shows the impact on the calculated MRC of `mcf` as a larger and larger percentage of its trace log entries are ignored. These artificial degradations to the trace log are indicated by labels such as "keep every 4^{th}", which simulates the impact of dropping 3 events and keeping the next event. The larger 1600k-entry trace log was used to ensure adequate trace log lengths. In order to see the impact on the vertical shifting of the curves, v-offset matching was not applied. Similar trends were seen for the other applications and are not shown here.

From these results, we can see that the v-offset of the calculated MRC is affected. As the number of events missed increases, the MRC is shifted further down. There is also a potential impact on the MRC shape, affecting the smaller cache sizes. The precise magnitude of the shifting or shape distortion varies across applications and shows no predictable pattern. By extrapolating these trends backwards, we can conclude that missed events are a potential source of the v-offset mismatch and shape distortion between the real and calculated MRCs.

Hypothetically, if the POWER5 PMU had been able to report the number of events lost during L2 cache access tracing, then it may have been possible to make the v-offset adjustments based on these values instead of the method described in Section 5.3.2.

Characterizing Trace Log Information Loss

To obtain an idea of how many events can be missing from the trace log and the characteristics of these missed events, we ran a microbenchmark with a well-known access pattern and very high access pressure to the L2 cache. The well-known access pattern enables us to compare the ideal trace log content to the actual content obtained via the POWER5 PMU. The very high access pressure to the L2 cache creates a worse-case scenario and represents a near upper-bound on the amount of information loss.

The C language source code of the microbenchmark is shown Figure 5.11. This microbenchmark linearly traverses an array continuously in an infinite loop. To maintain high L2 cache access pressure, the array size and array access stride were carefully chosen with four factors in mind. (1) TLB (translation look-aside buffer) misses can stall the processor pipeline, leading to decreased L2 cache access pressure. To prevent this problem from occurring, a relatively small array was used so as to not exceed the address-span of the POWER5 first-level data TLB. (2) The L1 data cache can prevent accesses from reaching the L2 cache, leading to decreased L2 cache access pressure. To prevent this problem from occurring, the array size was also chosen so that it was large enough to exceed the size of the L1 data cache and thus requires an L2 cache access for every array access. (3) To prevent the POWER5 L1 data cache hardware prefetcher from operating and interfering with our benchmark, we used an access stride length of 256 bytes. This stride length causes every other cache line of the array to be accessed, rather than every line, and thus prevents the prefetcher

```
#define ARRAYSIZE 131072  /* A 128 kB array (32 pages). */
#define STRIDE 256        /* To defeat the L1 data cache prefetcher. */
char array[ARRAYSIZE];

void main() {
    while (1) {
        int i;
        for (i = 0 ; i < ARRAYSIZE ; i = i + STRIDE) {
            char garbage = array[i];
        }
    }
}
```

Figure 5.11: C language source code of the trace log microbenchmark. It linearly traverses an array in a tight infinite loop.

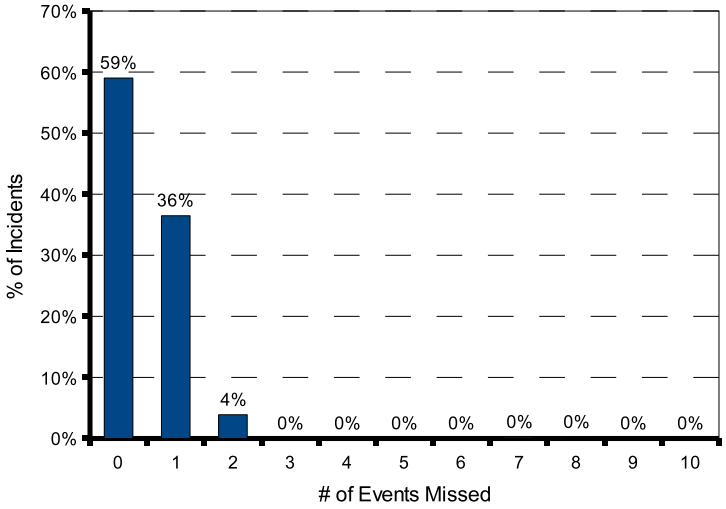


Figure 5.12: Histogram of missed events. The x-axis indicates the number of missed events between adjacent trace log entries. These results indicate that under a worse-case scenario, the POWER5 PMU is able to capture into its trace log all events 59% of the time and every other event 36% of the time.

from detecting an access pattern. (4) Finally, to keep the 2 load/store units of the POWER5 occupied and thus exerting full access pressure on the L2 cache, the array accesses have no data dependencies among each other and so they can proceed in parallel in the hardware. There is a risk that the C compiler (GCC 4.3) could optimize the given source code by eliminating the array accesses altogether since the local temporary `garbage` variable is never subsequently used. We verified that the compiled code, compiled without any optimization flags, does indeed perform the array read access, by inspecting the generated intermediate machine assembly code.

Figure 5.12 shows a histogram of the results. It characterizes the information loss by classifying the number of events missed between adjacent entries in the trace log. Overall, the histogram indicates that under a worse-case scenario, the POWER5 PMU is typically capable of capturing all events or every other event in its trace log. In more detail, the histogram shows that 59% of the trace log contains no missed events between adjacent entries, while 36% of the trace log is missing 1 event between adjacent entries, 4% of the trace log is missing 2 events, and the remaining 1% is distributed among the rest, ranging from 3 to 10 missed events.

A compact visualization of the trace log (first 6,130 entries) is shown in Figure 5.13. Each digit indicates the number of events missing between adjacent trace log entries. These digits are ordered from left to right, top to bottom, analogous to English language reading conventions. The figure indicates that, for example, between the 1^{st} and 2^{nd} entries of the trace log, 1 event is missing, while between the 2^{nd} and 3^{rd} entries, no events are missing. In fact, no events are missing from the 2^{nd} until the 9^{th} entries. Overall, the results show that there are long segments where no events are missing between trace log entries, and that there are also long segments where 1 event is missing between entries. The maximum number of events missed was found to be 9 events in our microbenchmark.

5.6.6 Impact of Set Associativity

In theory, direct-mapped and set-associative caches, if accessed uniformly, have the same hit/miss rate characteristics as a fully associative cache. In reality, applications do not access caches in such a manner, leading to cache set conflicts that cause higher miss rates. Many L2 caches on existing processors have high associativity, such as 10-way or 16-way in an attempt to mitigate the problem for moderately non-uniform access patterns. In addition, rather than implement a true LRU (least-recently-used) cache line replacement policy within each associative set, which is costly to implement in silicon, processors typically implement a pseudo-LRU policy, which closely approximates true LRU [Al-Zoubi et al. 2004]. For example, the POWER5 processor implements a pseudo-LRU policy known as pairwise-compare [Zhang et al. 2008].

To examine the impact of using a fully associative cache model compared against the 10-way set-associative cache used by the POWER5 processor, we fed our trace log into the Dinero cache simulator [Edler and Hill]. Given that pseudo-LRU replacement policies closely approximate a true LRU policy [Al-Zoubi et al. 2004], we configured Dinero to simulate a 1.875 MB L2 cache with a true LRU cache line replacement policy within each associative set. The associativity was varied from 10-way to full associativity, and the impact on the miss rate was extracted. The results for `mcf` are shown in Figure 5.10d. Similar trends were seen for the other applications. The graph indicates that our fully associative cache model simplification does not have a material impact on calculated cache miss rates.

5.6.7 Impact of Hardware Prefetching

Hardware prefetchers, located in the L1 data cache and the L2 cache can have an impact on both the real and calculated MRCs. In terms of the real MRCs, for some applications, the prefetchers can pollute the L2 cache and lead to a higher miss rate than without the prefetcher. On the other hand, the opposite can also occur. For other applications, the prefetchers can be beneficial and help overcome the problems of smaller cache sizes, resulting in lower miss rates.

In terms of the real MRCs, Figure 5.14a and Figure 5.14b show the impact on the real MRC of

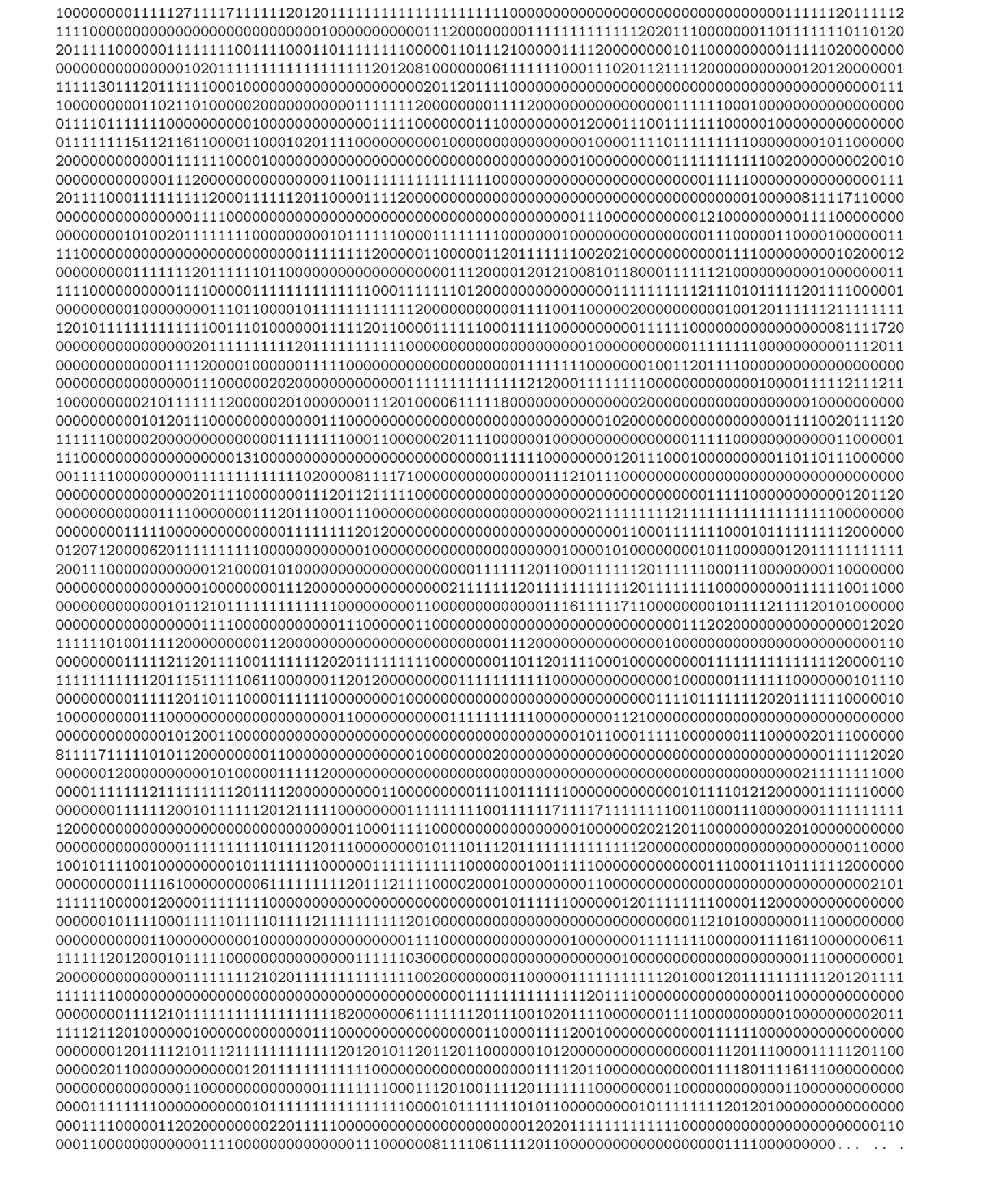
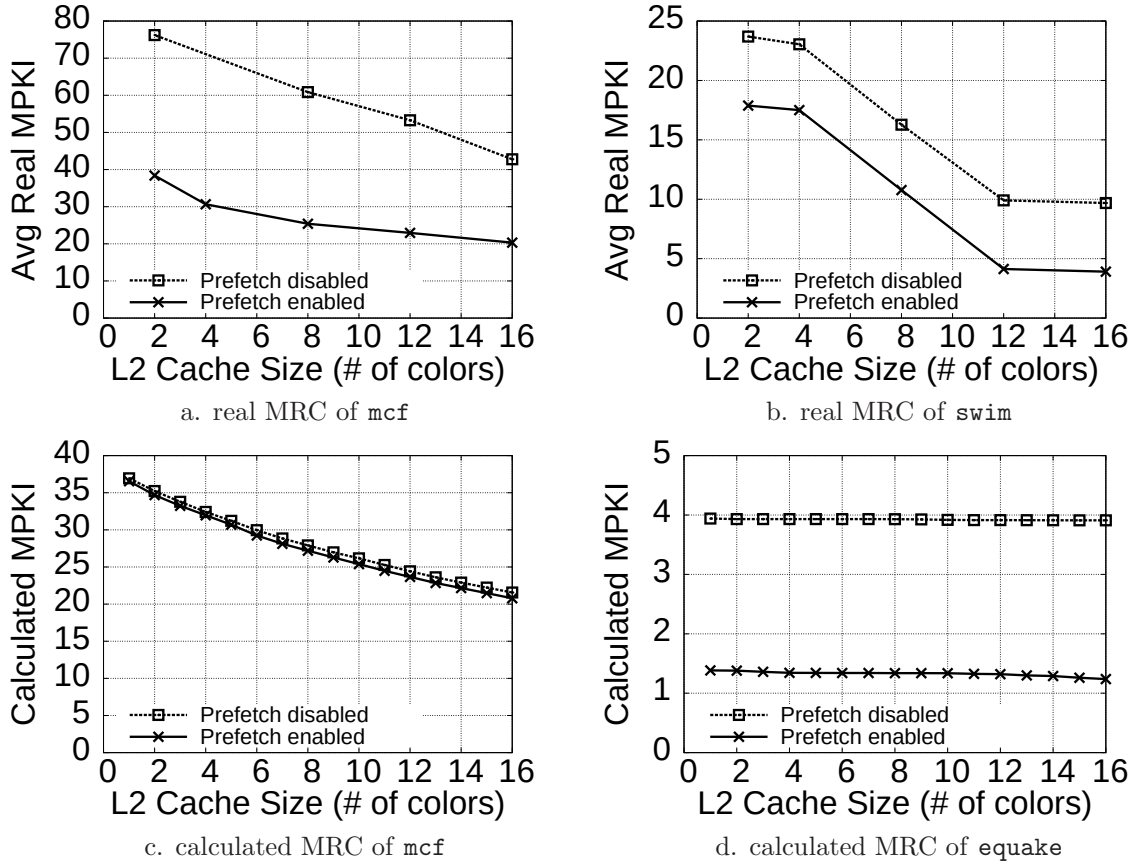


Figure 5.13: A compact visualization of the microbenchmark trace log (first 6,130 entries). Each digit indicates the number of events missing between adjacent trace log entries (max. value of 9). These digits are ordered from left to right, top to bottom. There are long segments where no events are missing between trace log entries, and there are also long segments where 1 event is missing between entries.



a. real MRC of `mcf`

b. real MRC of `swim`

c. calculated MRC of `mcf`

d. calculated MRC of `equake`

Figure 5.14: Impact of prefetching on real and calculated MRCs. (a)(b) Prefetching is beneficial to both of the real MRCs, vertically shifting the curves downward by varying amounts. (c)(d) Prefetching causes the calculated MRCs to be shifted downwards due to similar reasons as to when L2 cache access events are missed in the trace log.

`mcf` and `swim`, respectively, of disabling the hardware prefetchers on the POWER5+ system. The graphs indicate that the POWER5+ hardware prefetchers are beneficial to the workloads, helping to reduce the miss rate, vertically shifting the real MRC downwards compared to the configuration without prefetching. Similar trends were seen for 7 other applications that we tried[6].

In terms of the calculated MRCs, hardware prefetchers can have an impact on the captured trace log because prefetching is occurring during the capture period, thus also affecting the calculated MRC. On the POWER5 processor, these prefetched addresses appear in the trace log as a series of consecutive repeated data address values, as described in Section 5.3.1, but they do not show the actual values. As described in Section 5.3.1, we converted these repetitions into consecutive adjacent cache line addresses. Table 5.5, column e shows the percentage of the trace log that required this conversion. In contrast, the POWER5+ processor omits this prefetch activity altogether from the trace log. Therefore, both processors cannot provide adequate information to accurately model the prefetcher impact on the calculated MRC. In effect, this problem causes an increase in the number of events missing from our trace log, leading to the problems of vertical shifting and possible shape

[6] `applu`, `apsi`, `art`, `equake`, `mgrid`, `twolf`, `vpr`.

distortion of the calculated MRC as described in Section 5.6.5. To model prefetcher activity, we would need to capture all L1 data cache accesses, both hits and misses, and feed them into a prefetch simulator. However, this approach incurs extremely high overhead due to the amount of data required, making it impractical for online use.

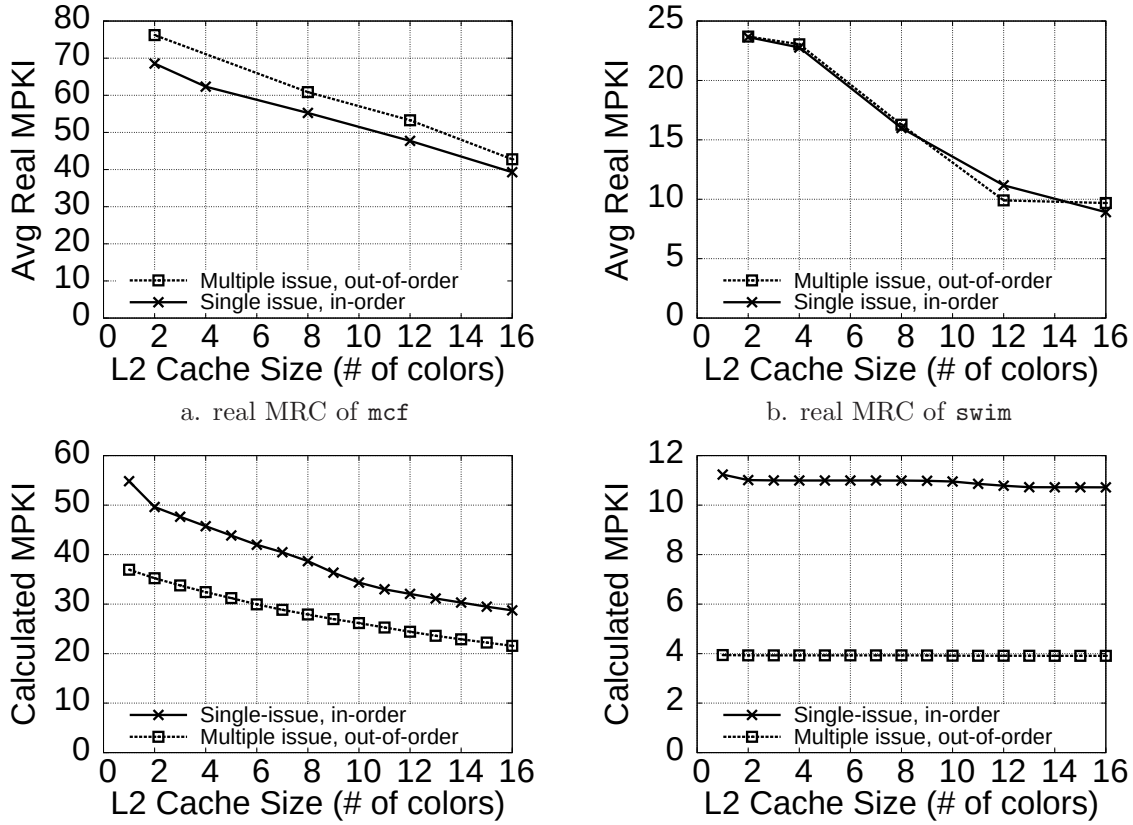
In an attempt to determine the impact of hardware prefetching on the calculated MRC, we compare the calculated MRC (RapidMRC) from a trace log obtained with prefetching against one obtained without prefetching. These experiments were run on the POWER5+ system. Figure 5.14c and Figure 5.14d show two examples of how these MRCs are affected for `mcf` and `equake`, respectively. The other applications show similar results. The graphs indicate that when prefetching is enabled, the calculate MRCs are vertically shifted downward compared to without prefetching, by different, currently unpredictable amounts, perhaps dependent on the application access pattern.

At first glance, one may expect that since the POWER5+ PMU completely ignores L2 cache access events due to prefetches, the trace log content should be unaffected by whether hardware prefetching is enabled or disabled. The trace log should look identical under either configurations, and should cause the two curves to be completely identical and overlapping. However, the results in Figure 5.14c and Figure 5.14d show the contrary. This difference may be due to the fact that these prefetches alter the content of the L1 data cache. Such alterations of the L1 data cache can lead to greater or fewer subsequent L1 data cache accesses, leading to different trace log content captured by RapidMRC, ultimately leading to different calculated MRCs. In Figure 5.14d, for example, when prefetching is enabled, the L1 data cache may be able to successfully service more requests than otherwise, leading to fewer accesses to the L2 cache. These fewer L2 cache accesses give the appearance that the trace log has missed capturing some of the L2 cache access events. As seen in the Section 5.6.5 on the impact of missed events, such missed events lead to vertical shifting of the calculated MRC downwards and potential curve shape distortion, which is the same pattern seen in Figure 5.14d.

Although we would like to model the prefetcher in our MRC calculation engine for improved accuracy, we are missing the additional access information required to model the prefetcher. We would need to capture all L1 data cache hits in addition to the misses that we are currently capturing, which is not a practical performance-accuracy trade-off for our online goals. The current practical solution may be to detect this prefetcher activity and conservatively emit a warning about the possibility of inaccuracy.

5.6.8 Impact of Multiple Instruction Issue & Out-of-Order Execution

Allowing multiple instructions to be concurrently issued and be in-flight in the processor pipeline may potentially lead to inaccuracies in RapidMRC due to missed events, as described Section 5.3.1. In this section, in order to eliminate any potential interference from the hardware prefetcher, hardware prefetching was disabled during miss rate capture for the real MRCs and during trace log capture for the calculated MRCs.



a. real MRC of `mcf`

b. real MRC of `swim`

c. calculated MRC of `mcf`

d. calculated MRC of `equake`

Figure 5.15: Impact of multiple instruction issue and out-of-order execution on real and calculated MRCs. (a)(b) They can vertically shift the real MRC upwards. (c)(d) They vertically shift the calculated MRCs downwards and cause shape distortion because they cause events to be missed in trace log.

In terms of the real MRC, Figure 5.15a and Figure 5.15b show the impact on the real MRC of `mcf` and `swim`, respectively, from multiple instruction issue and out-of-order execution. The graphs indicate that multiple instruction issue and out-of-order execution can shift the real MRC upwards. Similar trends were seen for 7 other applications that we tried[7].

In terms of the impact on the calculated MRC (RapidMRC), Figure 5.14c and Figure 5.14d show the impact on RapidMRC as calculated on the POWER5+ system for `mcf` and `equake`. For the curves labelled "single issue, in-order", the processor executes the application in complex mode (multiple instruction issue, out-of-order execution, hardware prefetching) except during the trace collection period when it is placed into the simplified mode (single instruction issue, in-order execution, no hardware prefetching). The graphs indicate that multiple instruction issue and out-of-order execution shift the calculated MRC downward by varying amounts, dependent upon the application. Note that in contrast, the real MRCs were shifted upwards. There is also a potential impact on the calculated MRC shape, affecting the smaller cache sizes. This impact on the calculated MRC is perhaps due to the fact that when multiple instruction issue and out-of-order execution are enabled, L2 cache access events can be missed in the trace log, as described in

[7] `applu`, `apsi`, `art`, `mgrid`, `swim`, `twolf`, `vpr`.

Section 5.3.1. The precise magnitude of the shifting or shape distortion varies across applications and shows no predictable pattern. Other applications had similar trends and are therefore not shown. As described in Section 5.6.1, `art` showed significant accuracy improvement in Figure 5.9b with the simplified processor mode.

5.7 RapidMRC for Provisioning the Shared Cache

We briefly evaluate the usefulness of RapidMRC by applying it to cache provisioning among multiprogrammed workloads running on a shared-cache multicore processor. We compare the partition size chosen using the MRC supplied by RapidMRC versus using the MRC supplied by the offline real MRC. Since our prototype implementation computes RapidMRC only once, we have selected applications that have fairly stable behaviour throughout the measurement period.

We ran experiments with the following pairs of applications: `twolf+equake`, `vpr+applu`, and `ammp+3applu`. `twolf+equake` and `vpr+applu`[8] were executed on the POWER5+ system but with the 36 MB L3 cache disabled because we found that the small working set size of these application pairs (from SPECcpu2000), combined with the abnormally large L3 cache, eliminated any shared cache performance problems: with the L3 cache enabled, the application pairs experienced a 98% hit rate to the L2 or L3 caches, leaving only 2% of accesses to main memory. Consequently, we disabled this unusually large L3 cache to re-introduce the shared cache performance problems seen by previous researchers who used commonly-found dual-core hardware configurations that do not contain L3 caches [Lin et al. 2008; Qureshi and Patt 2006; Zhang et al. 2009a].

The `ammp+3applu` workload, in contrast, demonstrates a fully utilized hardware configuration. It was run on the POWER5 system, utilizing the 36 MB L3 cache and all 4 SMT hardware contexts. To reduce its search space, all 3 instances of `applu` were confined to sharing the same cache partition[9].

Since the chosen applications exhibit fairly stable behaviour, the application MRCs shown in Figure 5.8 from both the RapidMRC and offline real MRCs were first fed as inputs to the *SizeSelection*() function in Equation 5.3. Although this algorithm can be executed online with low overhead, due to limitations in our prototype implementation of RapidMRC, we used this algorithm offline. The resulting chosen partition sizes are shown in the table of Figure 5.16.

Next, we ran the selected applications together with the L2 cache partitioned according to the suggested partition sizes. In addition to the cache configurations chosen from RapidMRC and the real MRC, all other possible partition sizes were also run to obtain an entire spectrum of multiprogrammed performance results, as shown in Figure 5.16. Graphs (a),(b), and (c) in Figure 5.16 show the average *individual* IPC of each application for the entire multiprogrammed

[8] Only the "place" phase of `vpr` was utilized.

[9] A simple heuristic is to place all cache-insensitive applications, indicated by their horizontally-flat RapidMRCs, into a single shared cache partition.

Chosen Cache Sizes		
Applications	**Real MRC** (#colors : #colors)	**RapidMRC** (#colors : #colors)
twolf : equake	14 : 2	9 : 7
vpr : applu	15 : 1	9 : 7
ammp : 3applu	13 : 3	14 : 2

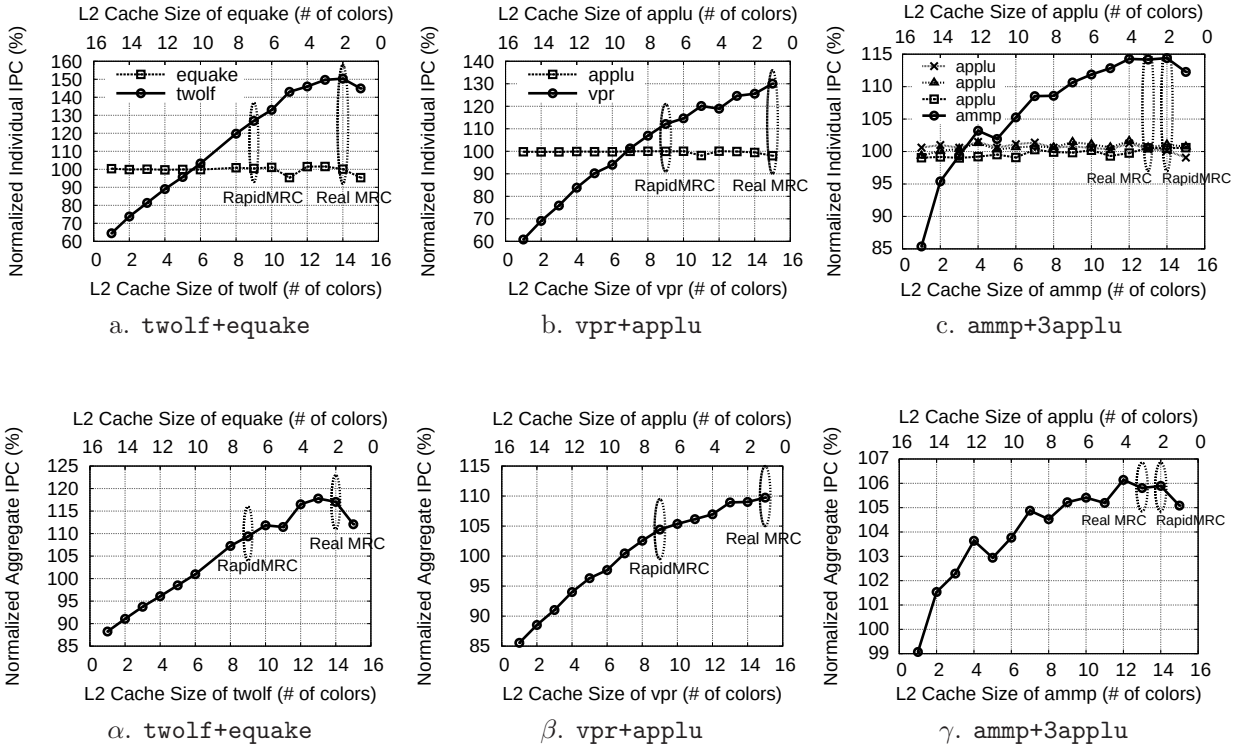


Figure 5.16: Chosen cache sizes and multiprogrammed workload performance as a function of L2 cache size. The performance of each individual application is normalized to its performance where both applications are running simultaneously but with uncontrolled sharing of the L2 cache. Graphs a, b, and c show the average *individual* IPC of each application while graphs α, β, and γ show the average *aggregate* (total system) IPC of the system. The bottom x-axis shows the number of colors (N) given to one application, while the remaining $16 - N$ colors are given to the second application, as indicated by the top x-axis. A vertical line draw at any point indicates the normalized performance of each of the two applications for a given partitioning of the L2 cache. The dashed ellipses in the graphs indicate the partition sizes selected using the MRCs from RapidMRC and the offline real MRCs, as indicated in the table.

run of the application, normalized to the uncontrolled sharing configuration. Graphs (α), (β), and (γ) in Figure 5.16 illustrate a different view of the same data, showing the average *aggregate* (total system) IPC of the applications for the entire multiprogrammed run, normalized to the uncontrolled sharing configuration. Aggregate IPC is calculated as follows, where $IPC(i)$ is the average IPC of the i^{th} application of a multiprogrammed workload consisting of N applications:

$$\frac{\sum_{i=0}^{N-1} IPC(i)_{new}}{\sum_{i=0}^{N-1} IPC(i)_{old}} \tag{5.5}$$

The multiprogrammed combinations are terminated as soon as one of the applications end.

In the graphs of Figure 5.16, the bottom x-axis shows the number of colors (N) given to one application, while the remaining $16 - N$ colors are given to the second application, as indicated by the top x-axis. Note that the 2 x-axes run in opposite directions so that a vertical line drawn at any point will indicate a total of 16 colors allocated among the two applications, meaning that the entire L2 cache is used. In graphs a, b, and c, the y-axis indicates performance, in terms of instructions-per-second (IPC), of each application, normalized to its performance where both applications are running simultaneously but with uncontrolled sharing of the L2 cache. In graphs α, β, and γ, the y-axis indicates the aggregate performance of the system. The y-axis performance values are the averages over the entire execution of the application pair, captured using a hardware performance counter window size of 1 billion cycles. A vertical line draw at any point in the graph indicates the normalized performance of the two applications for a given partitioning of the L2 cache. As a concrete example, in Figure 5.16a, when `twolf` is given 9 colors and `equake` is given the remaining 7 colors, the IPC of `twolf` increases by 27%, compared to when the L2 cache sharing is uncontrolled, while the IPC of `equake` remains largely unaffected.

The dashed ellipses in the graphs, labelled `RapidMRC` and `Real MRC`, indicate the partition sizes selected using the MRCs from RapidMRC and the offline real MRCs, respectively.

Graphs α, β, and γ indicate only modest improvements in total system performance, in terms of aggregate (total system) IPC, using RapidMRC curves (9%, 4%, and 6%, respectively) or offline L2 MRCs (17%, 10%, and 6%). However, viewed from a different perspective, the individual IPC of each application has been significantly improved. Graphs a, b, and c in Figure 5.16 show that using RapidMRC to provision the cache, `twolf` can improve by 27% without impacting `equake`, `vpr` can improve by 12% without impacting `applu`, and `ammp` can improve by 14% without impacting the 3 instances of `applu`. In contrast, using the sizes chosen using the offline real MRCs resulted in IPC improvements of 50%, 28%, and 14%, respectively. The cause of the performance gaps is due to the inaccuracies of the online calculated MRCs compared to the offline real MRCs. The horizontally flat sections of the calculated MRCs in `twolf` and `vpr`, seen in Figure 5.8, prevent the size selection algorithm from choosing the same sizes as those of the offline real MRCs. Despite the performance gaps, these results illustrate that the MRC supplied by RapidMRC can help achieve performance gains when applied to cache provisioning.

The differences between the *individual* IPC values and the *aggregate* (total system) IPC values

are purely due to the way in which they are calculated. For example, the 27% individual IPC improvement of `twolf` is significantly different from the 9% aggregate total system IPC improvement of its multiprogrammed workload combination. A simple, but similar, example using concrete numbers may help to illustrate such unintuitive mathematical outcomes. Given a multiprogrammed workload combination consisting of application A and B, if application A improves from an initial IPC value of 4 to a final value of 5, then its individual IPC improvement is 25%. If application B has an IPC value of 6 that does not change, then its individual IPC improvement is 0%. On the other hand, given these IPC values, the aggregate total system IPC has an initial value of 10 (4 from A + 6 from B) and a final value of 11 (5 from A + 6 from B), leading to only a 10% improvement in aggregate total system IPC.

For future work, we envision extending our prototype implementation to dynamically track MRC transitions, based on the light-weight method described in Section 5.6.2, and recompute optimal partition sizes accordingly. To enable dynamic L2 cache partition resizing in this vision, we have already implemented a page migration mechanism, as previously described in Chapter 4, with an attendant cost of 7.3 μs per 4 kB page.

5.8 Discussion

In developing RapidMRC, we have pushed the envelope of what is possible using today's PMUs. In particular, we have taken the data address capturing feature of the POWER5 processor, which is primarily intended for the purpose of *sampling*, to the extreme so that it can be used for *tracing*. To trace in this way, we force the processor to raise an exception at every L1 data cache miss, which obviously has substantial overhead. In addition, this method of tracing is inherently incomplete, as some data accesses are not recorded by the hardware PMU due to concurrency with other accesses. Fortunately, the results of our experimental analysis show that even with these limitations, the calculated MRCs are accurate in most cases, and the performance overhead is acceptable for our purposes. However, we believe more adequate hardware monitoring support will facilitate producing more accurate MRCs with much lower overhead.

Based on our experience, there are a few capabilities we would like to see in future PMUs. The first one is the ability of *tracing* data addresses into a small trace buffer, rather than a single data address register. This feature would allow an overflow exception to be raised only when the buffer is full, as opposed to on every data access. This would amortize the cost of exception handling over many data samples and thus greatly reduce monitoring overhead. A subset of these features can be found in existing processors, but not all of the features. For example, the Itanium 2 PMU and Intel x86 PEBS facility support a trace buffer. However, they are unable to operate at the fine time granularity required for tracing [Buck and Hollingsworth 2004; Lu et al. 2004; Marathe et al. 2005]. Additionally, the Intel x86 PEBS facility is unable to directly capture the data address [Sprunt 2002]. As for the second capability of future PMUs, the trace buffer should be capable of recording

all accesses, despite having several memory instructions in-flight. This seems to be feasible with a trace buffer instead of a single data address register. Thirdly, all accesses to the on-chip cache should be recordable, regardless of whether they are the result of processor memory instructions or hardware prefetchers. With these three features, for a short period of time, a complete trace of memory accesses performed by an application can be recorded.

5.9 Concluding Remarks

In this chapter, we have demonstrated a technique, called RapidMRC, to obtain the L2 miss rate curve of an application online by exploiting hardware PMUs and their associated hardware performance counters found in modern processors. We have also shown that our transparent method produces fairly accurate MRCs with a runtime overhead that is substantially less than other software-based approaches. In this dissertation, we utilize these calculated MRCs for cache provisioning, determining cache partition sizes in a shared cache environment, enabling up to 27% performance improvement compared to an uncontrolled cache sharing scheme. We believe that by providing a fairly accurate estimate of the cache needs of applications, RapidMRC will enable further optimization opportunities in on-chip caches.

We acknowledge the fact that we have exploited a PMU feature, i.e., continuous data address sampling, that is currently available only in IBM POWER5 processors. However, by demonstrating what can be accomplished with a simple PMU feature that is already implemented in an existing processor, we hope to provide motivation to other processor vendors to adopt similar PMU features.

Some possible future research directions include: (1) extending L2 MRCs to account for the impact of non-uniform miss latencies in addition to predicting the impact of misses on processor stall cycles, (2) exploring other online optimization opportunities that can be pursued using the online information provided by RapidMRC and, (3) exploring other online optimization opportunities that can be pursued using the underlying tracing mechanism. Additional details are provided in Chapter 7: Future Work.

Chapter 6

Discussion

"The voyage of discovery is not in seeking new landscapes but in having new eyes." – Marcel Proust

In this chapter, we discuss a number of topics relating to our research. We discuss:

1. alternative perspectives of what this dissertation demonstrates,
2. the value of hardware performance monitoring units (PMUs) in our work,
3. the role that additional hardware extensions in microprocessors could play,
4. the applicability of shared-cache management principles to other layers of the system stack,
5. the limitations of our work,
6. the shape of future systems and issues of scalability.

6.1 Alternative Perspectives

In the previous three chapters, we demonstrated the benefits of the two principles of managing a shared cache: promoting sharing and providing isolation. The two management principles attempt to maximize an advantage of shared caches (fast communications) and minimize a disadvantage of shared caches (interference).

From another perspective, this dissertation demonstrated how two major responsibilities of the operating system, scheduling and memory management, can be adapted for multicore processor systems. It is through these two responsibilities that the operating system can have a significant impact on application performance. We explored (1) how the operating system scheduler can promote shared use of the shared cache by an application, and (2) how the operating system memory manager can provide cache space isolation in the shared cache between disparate applications.

From a workloads perspective, this dissertation addressed both multithreaded and multiprogrammed computing environments. Multithreaded workload performance was improved via thread

clustering to promote the shared use of shared caches. Multiprogrammed workload performance was improved via cache partitioning to provide isolation, with optimal provisioning determined via the RapidMRC technique.

From a software-capabilities perspective, this dissertation can be viewed as pushing the envelope of what is possible on existing hardware with software-only techniques. In our investigative process, we have invalidated some assumptions of what is possible and not possible in terms of functionality, accuracy, utility, and overhead. For example, the software-based approach of RapidMRC, at first glance, appeared to be infeasible in all aspects: functionality, accuracy, utility, and overhead. In fact, prior work by other researchers have all proposed new hardware to pursue similar objectives. However, our experimental results have shown that the approach taken by RapidMRC is indeed feasible. In addition, our experience has constructively revealed where specific, incremental, minor improvements to existing hardware can be made, such as in the capabilities of existing hardware PMUs.

Finally, this dissertation provides an example of how new hardware developments induce new software-based research. In particular, our research was spawned from the development of (1) multicore processors and (2) advanced hardware PMU capabilities. The evolution of hardware provides a concrete driving force for re-evaluating existing software infrastructure and finding new opportunities for improvements, thus evolving the software.

6.2 The Value of Hardware Performance Monitoring Units

Our work has demonstrated several novel uses of hardware PMUs. PMUs were added to processors initially to provide low-level monitoring data for hardware designers, but we have demonstrated how PMUs can be exploited by software to become an effective component in a closely-coupled, fine-grained, timely feedback loop of a dynamic optimization system. In our context, the operating system is this dynamic optimization system since it makes management decisions at run-time in an effort to optimize performance.

Our research work would have been impossible without advanced hardware PMUs. They allowed us to dynamically monitor system performance and behaviour at a finer granularity, in terms of time and entity size, than would have been possible using software only, enabling the operating system to perform useful dynamic optimizations. In Chapter 3, they allowed us to monitor remote cache accesses in order to construct an online view of shared access to virtual memory regions for promoting shared use of the shared cache. In Chapter 5, they allowed us to monitor L2 cache accesses in order to construct L2 cache miss rate curves used in cache provisioning.

Hardware PMUs also allowed us to examine the impact of applications and the operating system on the hardware microarchitecture in an adequately detailed but post-mortem fashion. This ability allowed us to mostly bypass the need for full-system, performance-accurate simulators, such as Simics [Magnusson et al. 2002], to obtain detailed characteristics.

Despite these advantages, PMUs have capabilities, interfaces, and configuration settings that vary across processors and even within the same processor family, such as x86. This lack of standardization is a barrier to wide-spread adoption of PMUs by software developers. Standardization of hardware PMUs, which will be further described in Section 6.5.2, would enable hardware PMUs to be widely used by software in an effective and portable way.

6.3 Additional Hardware Extensions

New hardware extensions, as typically proposed by architecture researchers, are an alternative approach to our operating system-based solution to fulfilling the shared-cache management principles of promoting sharing and providing isolation. In addition, they can help in provisioning the shared cache. The benefits of a hardware solution may include lower runtime overheads and reduced code complexity of the operating system, however, this complexity burden would be merely shifted to the hardware design. Closer co-ordination between the hardware and the operating system would be required. No matter the implementation details, the need for promoting sharing, providing isolation, and provisioning remains.

In this dissertation, we showed how software-based techniques, with the help of existing hardware PMUs, can adequately fulfill the role of several previously proposed hardware extensions under several scenarios. The overheads, accuracy, and utility of software-based techniques were shown to be acceptable, reducing the urgency of incorporating new hardware extensions.

Software-based techniques have the major advantage that they can be physically deployed much quicker because they can be deployed on existing systems today. In contrast, hardware-based proposals have the disadvantage that they need to first be adopted by a major processor vendor and eventually shipped in future processors. The hurdles that need to be overcome in the adoption process are known to be onerous, reducing the probability of deployment.

6.4 Applicability to Other Layers of the System Stack

The shared-cache management principles demonstrated in this dissertation can be applied at various layers of the computer system software stack. In general, there is more domain-specific information at higher layers that can make the task simpler and more effective. However, applying the principles at a lower layer allows all higher-level components to benefit without any modifications, widening the scope of applicability. At a lower layer, there is a wider view of the computer system, enabling management policies to dynamically account for other applications, runtime systems, or operating systems that are running simultaneously alongside on the computer system.

6.4.1 Promoting Sharing in the Shared Cache

Promoting sharing via thread clustering could be applied at the application level and managed runtime system level. Existing operating systems, such as Linux, allow applications to configure the processor affinity of threads, if so desired. Thread clustering could be directly coded into an application, where application-specific information is easily available for decision making, potentially resulting in greater benefits than if implemented at the operating system level. For example, a database application may directly cluster threads that are accessing the same tables onto the same chip. Implementing thread clustering at the managed runtime system level, such as in a Java virtual machine runtime system, would be similar to our operating system level implementation. Finally, there may be opportunities to promote sharing within the operating system itself. For example, thread clustering may be applicable to the multiple kernel threads that are used to perform network packet processing from multiple network connections.

6.4.2 Providing Isolation in the Shared Cache

Providing cache space isolation can be applied at the application level, managed runtime level, operating system level, or virtual machine monitor level. For the layers above the operating system, an interface to the operating system can be provided for directing cache space isolation policies that are driven by the managed runtime system or the application. The policies for cache space isolation could be directly coded into the application, where application-specific information is easily available for policy decision-making, potentially resulting in greater benefits than an operating system-level policy. For example, Lee et al. have recently demonstrated exactly these kinds of benefits from database-specific knowledge of the amount of L2 cache used by different queries [Lee et al. 2009]. Implementing cache space isolation policies at the managed runtime level may have the advantage of object-level granularity cache placement, based on contextual information available in the garbage collection system of the runtime environment. Cache partitioning could be extensively applied to the operating system itself, such as providing an exclusive cache partition for virtual-to-physical page table entries, so as to prevent these performance critical entries from being evicted from the L2 cache by kernel and application threads, thereby speeding up page table lookups on translation look-aside buffer (TLB) misses, as observed by Soares et al. [Soares et al. 2008]. Finally, cache space isolation could be applied at the virtual machine monitor level to isolate the cache space among multiple operating systems.

While Chapter 4 demonstrated providing cache space isolation among multiple applications, Soares et al. demonstrated how to apply isolation within a single application, which is at a finer granularity [Soares et al. 2008]. Different memory regions of an application may exhibit different reuse distances, leading to different performance sensitivities. Providing cache space isolation among different memory regions can have performance benefits. Although Soares et al. demonstrated an automated online technique performed by the operating system at page granularity, it

could be implemented more precisely at the object granularity using an offline profiling technique combined with compiler manipulation of data layout, as described by Lu et al. [Lu et al. 2009].

In theory, with some amount of co-ordination and co-operation, cache space isolation can be applied in a hierarchical manner, crossing all layers of the system stack. From the bottom layer upwards, a virtual machine monitor can provision a certain amount of the on-chip shared L2 cache to each operating system. In turn, each operating system can further provision its allocation among its applications. Finally, the application can further provision its allocation among its memory regions. However, this hierarchical partitioning would require having many more partitions available in the cache, such as the 64 partitions possible on Intel Xeon multicore processors [Lin et al. 2008].

6.4.3 Provisioning the Shared Cache

In support of the principle of providing cache space isolation, our RapidMRC cache provisioning technique can be applied to obtain the cache requirements of various layers of the system stack, such as the cache requirements of an application, a managed run-time system, or an operating system running on top of a virtual machine monitor.

6.5 Limitations

We first discuss the issue of cache bandwidth management, followed by the limited applicability and portability of our techniques to other processors due to our dependence on the capabilities of the IBM POWER5 hardware PMU.

6.5.1 Cache Bandwidth Management

In this dissertation, we have dealt with managing the cache *space* of on-chip shared caches, but we have not dealt with managing the access *bandwidth* of the shared cache [Liu et al. 2010; Srikantaiah and Kandemir 2010]. For example, it could be advantageous to allocate 25% of the L2 cache bandwidth to one application and the remaining 75% to another application. Due to the fact that off-chip memory accesses are typically an order of magnitude slower than on-chip cache access, cache space issues are therefore more performance critical than cache access bandwidth issues. In addition, directly controlling the L2 cache access bandwidth among multiple applications or within a single application is not possible on existing multicore processors. However, L2 cache access bandwidth could be indirectly controlled by predicting bandwidth trade-offs and carefully co-scheduling applications accordingly to control or limit bandwidth consumption. Such an approach is similar to that taken by Antonopoulos et al. on traditional SMP multiprocessors to manage bus bandwidth [Antonopoulos et al. 2003]. Fedorova et al.'s approach, of compensative scheduling on multicore processors to compensate for cache interference, may have applicability in compensating applications that have had their L2 cache bandwidth victimized [Fedorova et al. 2007]. Zhang et

al.'s technique of duty-cycle modulation could be applied for this purpose as well [Zhang et al. 2009b].

6.5.2 Hardware Performance Monitoring Unit Capabilities and Portability

Our experiments were conducted on IBM POWER5 multicore processor systems, which contain advanced hardware PMU capabilities. Unfortunately, the POWER5 PMU features of data sampling and continuous data sampling are not generally available on the more mainstream Intel/AMD x86 processors at this time, which may make our techniques more complicated to implement on such systems.

Data sampling was essential to identifying and promoting sharing via thread clustering. Efficient data sampling is not widely and uniformly available on current x86 processors, although more recently available AMD processors are now capable of instruction-based data sampling, potentially allowing our thread clustering technique to be applied on these processors. Intel x86 processors do not yet directly support data sampling, but instead provide a capability known as precise event-based sampling (PEBS), which captures the entire architectural state of the processor upon a specified hardware event rather than capturing just the desired data item [Sprunt 2002]. It may be possible to construct a crude data sampling mechanism using PEBS, with a varying degree of implementation complexity, overhead, accuracy, and data loss. However, since PEBS captures a significant amount of state on each event, its overhead is significantly higher, and the granularity at which monitoring can occur is much coarser. Intel Itanium 2 processors, on the other hand, can directly perform data sampling [Buck and Hollingsworth 2004; Lu et al. 2004; Marathe and Mueller 2006].

Continuous data sampling was essential to obtaining a trace of L2 cache accesses in order to provision the shared cache via RapidMRC. RapidMRC requires the hardware PMU to be able to capture nearly every L2 cache access for a short period of time. Intel x86 hardware PMUs may be able to provide this information but with a certain degree of information loss via their precise event-based sampling (PEBS) mechanism. Researchers have also illustrated this problem on Intel Itanium 2 processors [Buck and Hollingsworth 2004; Marathe et al. 2005].

Our experience with varying hardware PMU capabilities, even within the same family of processors, leads us to believe that they should be standardized in an implementation-independent fashion so that they can be widely adopted across different platforms, perhaps analogous to the IEEE 754 floating-point standard [IEEE 2008]. In contrast to the well-established and stable instruction-set architecture (ISA) of processors, such as the x86 and PowerPC ISA specification, hardware PMU capabilities, interfaces, and configuration settings vary across processors and even within the *same* ISA family. For example, although the IBM PowerPC 970FX and IBM POWER5 processors both adhere to the PowerPC ISA specification, their hardware PMUs have different interfaces, a different set of capabilities, and different configuration settings. In order to comprehend and configure their respective hardware PMUs, the hardware PMU manuals of the specific processor must be carefully

read and understood. Worse, some of the functionality is not documented at all, and must be reverse engineered, perhaps because PMUs were primarily intended to be used by hardware engineers designing chips. The same problem arises for the x86 family of processors when comparing the hardware PMU of the Intel Core 2 line of processors versus the Intel Netburst line versus the AMD Opteron line.

Although software-based standardizations have been proposed, such as the Perfmon2 interface, they are missing a key piece of the puzzle [Eranian 2006]. Software-based standardizations can provide a fairly uniform interface and configuration settings, but it is beyond their mandate to advocate that a standard set of inherent hardware monitoring capabilities be available on all future processors. For example, it is beyond their scope to advocate that a data address tracing mechanism be provided in all PMUs. In addition, there may be overheads related to emulating, in software, missing hardware monitoring functionality, thereby reducing its attractiveness for online optimization usage. This problem may be analogous to the performance penalties experienced when attempting to emulate IEEE double-precision floating-point operations via a software library due to the lack of direct hardware support [IEEE 2008].

6.6 Future Systems and Scalability

Future systems software research directions will be largely influenced by the evolution of hardware. Future hardware will very likely consist of multiple larger scale processor chips, in terms of (1) the number of cores, (2) the number, size, and degree of sharing of on-chip caches, and (3) the complexity of the on-chip interconnection network [Huh et al. 2001; Kumar et al. 2005]. In addition, more heterogeneity and non-uniformity within a chip is very likely, in terms of (1) the functional capability, power consumption, and speed of cores, and (2) access latency, degree of sharing, and size of various levels of the cache hierarchy [Huh et al. 2005; Kumar et al. 2004].

Our work would be applicable to future heterogeneous multicore processors with shared on-chip caches, since we are only concerned with the shared caches and not the cores themselves. The uniformity or non-uniformity of the cores is an orthogonal issue. Researchers have already begun investigating operating system modifications that will be necessary on future heterogeneous multicores in order to extract the full performance potential of such anticipated hardware [Bower et al. 2008; Li et al. 2007; Shelepov et al. 2009; Suleman et al. 2009].

Non-uniformity on larger scale systems will mean that maximizing locality will be even more important to achieve desired performance in a scalable way. Veteran systems software researchers may experience a sense of *déjà vu*, given the similarities to performance scalability research that they may have conducted on traditional, unicore, large-scale NUMA multiprocessor systems. The once seemingly esoteric lessons learned about performance scalability on large-scale multiprocessors, such as the principles of maximizing locality and concurrency, are finally becoming widely relevant and increasingly important in the era of multicore processor systems [Gamsa 1999; Gamsa et al.

1999].

In fact, the past goals of the University of Toronto Hurricane/Tornado/K42 Operating System Group, of systems software performance scalability on large multiprocessors, have had a strong influence on the inception and research path of this dissertation. The underlying theme that is common to both Hurricane/Tornado/K42 and this dissertation is the examination of how operating systems can be designed and implemented to fully exploit the underlying hardware. Although we have not directly targetted performance scalability in this dissertation, our ideas, techniques, designs, and implementations serve as a precursor to scalability research on multicore processor systems. Having gained experience in extracting performance from small scale multicore processor systems, we are now in a better position to explore how to maintain these performance gains as these systems grow in size.

Thus, this dissertation can be viewed as having various direct and indirect contributions to the realm of scalability. Our exploration of promoting sharing via thread clustering can be viewed as being directly applicable to scalability research, by promoting locality to reduce cross-chip traffic via beneficial shared use of the on-chip shared caches. Providing flexible cache space isolation via partitioning can be viewed as enabling a greater number of cores to share a cache without interfering with each other's cache space, thus improving a small aspect of scalability. Finally, our provisioning technique is independent of the number of applications or cores, the size of caches, the number of distinct page colors and corresponding number of possible cache partitions, and the number of threads, meaning that it remains applicable on larger scale systems.

A new capability that was not available to traditional scalability researchers are the advanced hardware PMUs that are now found in multicore processors. In such complex computer systems, these advanced hardware PMUs will play an increasingly important role in monitoring and understanding the increasingly complex interactions between applications, systems software and hardware, enabling both offline and online scalability optimizations.

Chapter 7

Future Work

"The end is the beginning is the end." – William Patrick Corgan

There is much future work that could be done. In general, additional experiments could be run to strengthen the versatility, magnitude, and completeness of the results. Some additional experiments that are possible are: (1) studies with additional workloads, (2) the tuning and improving of various components of the developed mechanisms and policies, and (3) the integration of all mechanisms and policies developed in this dissertation into a single complete system that accounts for additional *real-world* factors, rather than having separate, *proof-of-concept* prototype implementations. Finally, we describe the foundation that has been built and that could be used for future research endeavors.

7.1 Running More Workloads

In general, more workloads, in the form of different application types and sizes could be run in conjunction with various hardware system sizes (containing more cores per chip).

For promoting sharing via thread clustering, additional multithreaded server workloads might include SPECjbb2005, SPECjAppServer, the TPC suite of database workloads, and game server workloads. In terms of size, experiments could be run with larger workload sizes on larger systems containing a greater number of processor chips. Greater benefits from larger systems are expected because the thread clustering technique of promoting sharing reduces cross-chip traffic on the shared system bus.

For providing isolation via cache partitioning, additional workloads might include more SPECcpu applications, various desktop applications, and server application workloads. Larger multicore processors containing more cores sharing an on-chip cache could be used to examine the benefits of isolation in the presence of greater interference potential.

For shared cache provisioning via the RapidMRC technique, additional workloads might include more SPECcpu applications, various desktop applications, and server application workloads. Processors containing larger on-chip L2 caches could be used to increase the cache size spectrum of the

calculated and real MRCs. The effectiveness of cache provisioning for more than 2 applications on larger systems can be further investigated by feeding the calculated MRCs (more than 2 of them) into Qureshi's approximation algorithm to determine partition size [Qureshi and Patt 2006].

7.2 Tuning and Improving Components

A number of components that we developed could be tuned or improved in order to yield potentially greater benefits. For promoting sharing via the thread clustering technique, the simple, one-pass clustering heuristic could be substituted with a heavier-weight, sophisticated, exhaustive formal clustering algorithm. The relatively simple similarity metric that we used could be substituted with more sophisticated ones.

For providing isolation via the cache partitioning technique, the dynamic partitioning aspects could be further investigated. For example, for dynamic partitioning re-sizing, it is necessary to copy the contents in the physical pages of one color to other colors. Precisely how the other remaining valid colors should be chosen as the copy destination could be further investigated, exploring if further performance improvements could be gained by more intelligent color selection. Another component that could be investigated, which is particular to the Linux operating system, is precisely how free physical pages should be obtained from the Linux global buddy allocator when there are no available local free physical pages.

For provisioning via the RapidMRC technique, the parameters explored for offline and potential online phase detection could be further investigated since we used only an arbitrarily chosen but fixed set of values. Given the online calculated MRCs, a variety of size-selection functions could be explored for selecting partition sizes among competing applications, as we only used the function that minimized total system miss rate. Due to hardware restrictions, we were unable to measure the exact number of events missed by the hardware PMUs during the logging period for our applications. Running additional microbenchmarks, similar to the one in Section 5.6.5, to determine the precise tracing capabilities of the hardware PMU might provide additional insights. Three out of the 30 applications showed unacceptable inaccuracies in their calculated versus real MRCs. Finding the root cause of these inaccuracies should be subject to further investigation. Finally, extending the MRC model to account for the impact of non-uniform miss latencies in addition to predicting the impact of misses on processor stall cycles could be investigated.

7.3 Further Systems Integration

Our implementations in this dissertation provided proof-of-concept prototypes that do not contain fully integrated solutions that address all potential problems stemming from our ideas.

The thread clustering technique interferes to some degree with the load-balancing goals of a traditional operating system scheduler. Further integration and modification of the load-balancing

system is needed to complement the thread clustering technique. For example, load balancing within a multicore chip should be performed only after thread clustering has determined how threads should be grouped. In addition, there may be scenarios when thread clustering must yield to the traditional load-balancing mechanism.

Our investigation into providing isolation via cache partitioning did not consider the additional possibility of time-multiplexing the shared cache among numerous competing applications. Cache partitioning should be closely tied to the operating system scheduling subsystem. The scheduler should attempt to co-schedule applications that do not occupy the same regions of the cache. For example, consider the scenario where application `A` is given the top 50% of the shared cache, application `B` is given the bottom 50% of the shared cache, and application `C` is also given the bottom 50% of the shared cache. An integrated scheduler may seldom co-schedule application `B` and application `C` together because they would otherwise induce cache space interference.

Cache provisioning via RapidMRC was shown in a component-wise fashion in this dissertation. Integrating the RapidMRC components into a complete runtime system is subject to future work. With a fully integrated running system, detailed overhead measurements would be possible. One necessary component is to integrate dynamic phase tracking based on our post-mortem but light-weight phase tracking method described in Section 5.6.2.

7.4 Enabling Future Research

Although we demonstrated the application of the shared-cache management principles at the operating system level, to the benefit of applications, these principles can be applied at various layers of the system software stack, as previously described in Section 6.4.

We have developed three mechanisms in this dissertation: (1) an online mechanism to promote sharing by detecting shared access to memory regions, as demonstrated by the thread clustering technique; (2) an online mechanism to control data placement in the L2 cache, as demonstrated by the software-based cache partitioning technique; and (3) an online mechanism to trace low-level, fine-grained hardware events, as demonstrated by the RapidMRC technique by the tracing of L2 cache accesses. These three mechanisms create further research possibilities that were not available in the past. Fellow researchers can use these initial mechanisms and ideas as a foundation or starting point for their own research, either further using, improving, and extending the base mechanisms, or to spawn or inspire related ideas. A few corresponding examples of future research possibilities include: (1) using the online sharing detection mechanism to improve locality and scalability on NUMA systems; (2) using the mechanism of online control of data placement in the L2 cache to mitigate cache set conflicts, detected with hardware PMUs; and (3) using the online fine-grained tracing mechanism to trace TLB (translation look-aside buffer) misses and improve physical page memory management algorithms [Azimi et al. 2007; Walsh 2009].

In particular, our RapidMRC technique, in addition to provisioning shared caches, could be

used in several other ways online: (i) reducing energy consumption by reducing the cache to the minimal size at which the running process/workload can still run effectively [Albonesi 1999; Balasubramonian et al. 2000; Meng et al. 2008]; (ii) managing bus bandwidth contention to main memory due to cache misses [Antonopoulos et al. 2003; Iyer et al. 2007]; (iii) guiding co-scheduling algorithms in selecting processes that fit within the available L2 cache space [Fedorova et al. 2005; Settle et al. 2004; Snavely and Tullsen 2000; Tam et al. 2007b]; (iv) predicting the global MRC of N applications in an uncontrolled cache-sharing configuration [Berg et al. 2006; Chandra et al. 2005]; and (v) identifying applications with low cache reuse so that they can all be placed into a single, shared *pollute buffer* cache [Soares et al. 2008].

Chapter 8

Conclusions

"One day, in retrospect, the years of struggle will strike you as the most beautiful." – Sigmund Freud

Our work has demonstrated that the operating system can effectively manage on-chip shared caches of multicore processor systems. We demonstrated that software-based techniques for managing the cache are feasible with the help of processor-embedded hardware performance monitoring units (PMUs). Rather than propose yet more extensions to hardware, we show feasible software-based approaches that are implementable today. The three key messages to *take away* from this dissertation are: (1) there are benefits in having the operating system manage the on-chip shared cache; (2) hardware PMUs and their associated hardware performance counters can play an important role in operating system online optimizations; and (3) operating systems must continually evolve as hardware evolves in order to fully exploit their beneficial characteristics and minimize their drawbacks.

The contributions that stem from our research are demonstrated by the three publications generated from this thesis. To the best of our knowledge, we are the first, on commodity multicore systems, to develop and demonstrate a methodology for the operating system to:

1. promote sharing by clustering sharing threads based on runtime information obtained using hardware PMUs [Tam et al. 2007b];

   On 3 multithreaded commercial server workloads from the SPECjbb2000, RUBiS, and VolanoMark benchmark suites, we were able to experimentally demonstrate peak performance improvements of 7%, due to a 70% reduction in processor pipeline stalls caused by cross-chip cache accesses, running on a 2-chip IBM POWER5 multicore system.

   A 14% potential performance improvement was experimentally demonstrated for SPECjbb2000 running on a larger-scale 8-chip IBM POWER5+ multicore system.

2. provide isolation by controlling occupation in the shared cache using a software-only technique [Tam et al. 2007a];

On 7 multiprogrammed workloads consisting of applications from the SPECcpu2000 and SPECjbb2000 benchmark suites, we were able to experimentally demonstrate peak performance improvements of 17%, in terms of instructions-per-cycle (IPC), running on an IBM POWER5 multicore system. In subsequent experiments, we were able to experimentally demonstrate peak performance improvements of 50%, in terms of IPC, when we disabled the abnormally large and fast off-chip L3 cache on an IBM POWER5+ multicore system in order to mimic more commonly found dual-core processors at the time of the experiments [Tam et al. 2009].

3. provision the shared cache by approximating L2 cache miss rate curves (MRCs) online using hardware PMUs [Tam et al. 2009];

   On 30 applications from the SPECcpu2006, SPECcpu2000, and SPECjbb2000 benchmark suites, we were able to experimentally demonstrate and illustrate the accuracy of these approximated MRCs.

   In addition, on 3 multiprogrammed workloads consisting of applications from the SPECcpu2000 benchmark suite, we were able to experimentally demonstrate that the accuracy of the approximated MRCs is adequate for cache provisioning, achieving peak performance improvements of 27%, in terms of IPC, running on an IBM POWER5+ multicore system.

In each case, we developed a new mechanism and provided experimental evidence that it can be used to achieve performance gains. In two of our contributions, we also specifically contribute to the hardware PMU research community by concretely demonstrating profitable online usage cases of specific hardware PMU features [Azimi et al. 2009]. The three mechanisms developed as a part of our work create further research possibilities. Fellow researchers can use these initial mechanisms and ideas as a foundation or starting point for their own work, either further using, improving, and extending the base mechanisms, or to spawn or inspire related ideas. We hope we have opened up new research possibilities to the research community.

A number of lessons were learned over the course of our research. Typically, these lessons stem from counter-intuitive expectations or unanticipated benefits that were not obvious at the onset of our research but only became obvious afterwards.

1. New hardware developments can be a good source for new software research. New hardware features, characteristics, and capabilities can make previously infeasible research ideas now feasible. Previous software design trade-offs were made based on previous hardware characteristics and these trade-offs have now changed, opening up new avenues of exploration. For example, our thread clustering technique was only feasible with the recent advancements in hardware PMUs.

2. The overhead and accuracy of software-based techniques, with the help of hardware PMUs, can be acceptable under certain scenarios. Software-based techniques have attractive trade-

offs compared to proposed future hardware-based techniques. For example, despite some of the limitations of our software-based cache partitioning technique, used to provide isolation, companies such as VMware have shown signs of interest, perhaps applying this software-only technique, which is widely applicable on nearly all existing hardware, to their virtual machine monitor software.

3. Seemingly infeasible ideas about how to exploit hardware PMUs need to be experimentally pursued on real hardware in order to see if they are indeed infeasible, in terms of functionally, accuracy, and overhead. Sometimes, these ideas turn out to be feasible, demonstrated only by experimentation on real hardware. The RapidMRC technique for provisioning is an example of such a seemingly infeasible idea.

4. Restricting the scope of research to real systems can have unexpected benefits. Although using real hardware can restrict some research possibilities, it can cause researchers to be creative along a different dimension. Rather than dreaming up what to do with extra transistors, as typically pursued by architecture researchers, it causes systems software researchers to push the envelope of what is possible with the given restrictions, squeezing every last bit of capability out of the existing hardware. This approach to research can lead to finding and re-defining new capabilities and trade-offs between software-based and hardware-based techniques.

5. Standardization of hardware PMUs is an important long term goal. In our work, we recognize that it will be a challenge to port our mechanisms and techniques to other processors, such as the x86 family, due to varying capabilities across and even within processor families. Having a universal hardware standard for hardware PMUs, perhaps analogous to the IEEE 754 floating-point standard, would guarantee that hardware PMU features, interfaces, and configuration details remain universally available and consistent, thus encouraging their wide-spread adoption by software developers for offline and online optimizations.

Bibliography

J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler. Technical report, Silicon Graphics, Inc., Feb. 17, 2005.

H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM Southeast Regional Conference (ACMSE)*, pages 267–272, 2004.

D. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 248–259, 1999.

AMD. New six-core AMD Opteron processor delivers up to thirty-four percent more performance-per-watt in exact same platform: Available five months ahead of schedule, "Istanbul" is an industry game-changer. AMD Press Release, Sunnyvale, CA, Jun. 1, 2009.

AMD. AMD introduces the world's most advanced x86 processor, designed for the demanding datacenter. AMD Press Release, Sunnyvale, CA, Sept. 10, 2007.

AMD. AMD announces world's first 64-bit, x86 multi-core processors for servers and workstations at second-anniversary celebration of AMD Opteron processor. AMD Press Release, New York, NY, Apr. 21, 2005.

G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R.W. Wisniewski. Libra: a library operating system for a JVM in a virtualized execution environment. In *Int'l Conf. on Virtual Execution Environments (VEE)*, pages 44–54, 2007.

C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2): 18–28, Feb. 1996.

J.H. Anderson, J.M. Calandrino, and U.C. Devi. Real-time scheduling on multicore platforms. In *Real-Time & Embedded Technology & Applications Symp. (RTAS)*, pages 179–190, 2006.

J.P. Anderson, S.A. Hoffman, J. Shifman, and R.J. Williams. D825 - a multiple-computer system for command & control. In *AFIPS Fall Joint Computer Conference (FJCC)*, pages 86–96, 1962.

T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel & Distributed Systems*, 1(1):6–16, Jan. 1990.

T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 108–120, 1991.

C.D. Antonopoulos, D.S. Nikolopoulos, and T. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *Int'l Conf. on Parallel Processing (ICPP)*, pages 547–554, 2003.

J. Appavoo. *Clustered Objects*. PhD thesis, Dept. of Computer Science, Univ. of Toronto, 2005.

J. Appavoo, D. Da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R.W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS)*, 25(3):6:1–6:52, Aug. 2007.

Apple. Apple introduces Power Mac G5 Quad & Power Mac G5 Dual. Apple Press Release, Cupertino, CA, Oct. 19, 2005.

M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 250–261, 2009.

R. Azimi. *System Software Utilization of Hardware Performance Monitoring Information*. PhD thesis, Dept. of Electrical & Computer Engineering, Univ. of Toronto, 2007.

R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Int'l Conf. on Supercomputing (ICS)*, pages 101–110, 2005.

R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. Demke Brown. PATH: Page access tracking to improve memory management. In *Int'l Symp. on Memory Management (ISMM)*, pages 31–42, 2007.

R. Azimi, D.K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM Special Interest Group in Operating Systems (SIGOPS), Operating Systems Review (OSR)*, 43(2):56–65, Apr. 2009.

M.J. Bach and S.J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical J.*, 63(8):1733–1749, Oct. 1984.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Conf. on Programming Language Design & Implementation (PLDI)*, pages 1–12, 2000.

R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 245–257, 2000.

L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 282–293, 2000.

A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Symp. on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

B.M. Beckmann and D.A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 319–330, 2004.

B.M. Beckmann, M.R. Marty, and D.A. Wood. ASR: Adaptive selective replication for CMP caches. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 443–454, 2006.

L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.*, 5(2):78–101, 1966.

F. Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. In *Symp. on Operating Systems Principles (SOSP) - Work in Progress Session*, 1997.

F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *J. of Parallel & Distributed Computing*, 37(1):113–121, Aug. 1996.

E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Int'l Symp. on Performance Analysis of Systems & Software (ISPASS)*, pages 20–27, 2004.

E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 169–180, 2005.

E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *Int'l Symp. on Performance Analysis of Systems & Software (ISPASS)*, pages 89–99, 2006.

B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 158–170, 1994.

M. Bhadauria, R. Huang, and S.A. McKee. Improving performance and energy through multithread program scheduling. Technical Report CSL-TR-2008-1053, Cornell Univ., Sept. 2008.

D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. Sciver, and P. Wang. OSF/1 virtual memory improvements. In *USENIX Mac Symp.*, pages 87–103, 1991.

S. Blagodurov, S. Zhuravlev, S. Lansiquot, and A. Fedorova. Addressing cache contention in multicore processors via scheduling. Technical Report TR 2009-16, Simon Fraser Univ., Jul. 2009.

G.E. Blelloch, R.A. Chowdhury, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Symp. on Discrete Algorithms (SODA)*, pages 501–510, 2008.

W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Symp. on Operating Systems Principles (SOSP)*, pages 19–31, 1989.

F.A. Bower, D.J. Sorin, and L.P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, May-Jun. 2008.

S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 43–57, 2008.

S. Boyd-Wickizer, R. Morris, and F. Kaashoek. Reinventing scheduling for multicore systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

Broadcom. Broadcom delivers industry's highest performance 64-bit MIPS-based, multi-processor solution for broadband networking. Broadcom Press Release, Irvine, CA, Jul. 18, 2001.

D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *Workshop on Feedback-Directed & Dynamic Optimization (FDDO)*, 2001.

B. Buck and J. Hollingsworth. A new hardware monitor design to measure data structure-specific cache eviction information. *J. of High Performance Computing Applications*, 20(6):353–363, Fall 2006.

B. Buck and J. Hollingsworth. Data centric cache measurement on the Intel Itanium 2 processor. In *Conf. on Supercomputing (SC)*, 2004.

B. Buck and J. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Conf. on Supercomputing (SC)*, 2000a.

B. Buck and J. K. Hollingsworth. An API for runtime code patching. *J. of High Performance Computing Applications*, 14(4):317–329, Winter 2000b.

J.R. Bulpin and I.A. Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conf.*, pages 103–106, 2005.

D. Burger, J.R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 78–89, 1996.

J. Burns and J.-L. Gaudiot. SMT layout overhead and scalability. *IEEE Transactions on Parallel & Distributed Systems*, 13(2):142–155, Feb. 2002.

R.W. Carr and J.L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. *ACM Special Interest Group in Operating Systems (SIGOPS), Operating Systems Review (OSR)*, 15(5):87–95, Dec. 1981.

C. Cascaval, E. Duesterwald, P.F. Sweeney, and R.W. Wisniewski. Performance and environment monitoring for Continuous Program Optimization. *IBM J. of Research & Development*, 50(2/3): 239–248, Mar. 2006.

K. Chan. An adaptive software transactional memory support for multi-core programming. Master's thesis, Univ. of Hong Kong, 2009.

D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Int'l Conf. on Supercomputing (ICS)*, pages 242–252, 2007.

J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 264–276, 2006.

J. Chapin, A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Int'l Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, 1995.

J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, Nov. 1994.

M. Chaudhuri. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 401–412, 2009.

J.B. Chen and B.N. Bershad. The impact of operating system structure on memory system performance. In *Symp. on Operating Systems Principles (SOSP)*, pages 120–133, 1993.

S. Chen, P.B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Symp. on Parallelism in Algorithms & Architectures (SPAA)*, pages 105–115, 2007.

T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers (TC)*, 44(5):609–623, May 1995.

C.-Y. Cher, A.L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 199–210, 2004.

Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 357–368, 2005.

S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 455–468, 2006.

S. Cho and L. Jin. Better than the two: Exceeding private and shared caches via two-dimensional page coloring. In *Workshop on Chip Multiprocessor Memory Systems & Interconnects (CMP-MSI)*, 2007.

J. Choi, S.H. Noh, S.L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 286–295, 2000.

L. Codrescu, D.S. Wills, and J. Meindl. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computers (TC)*, 50(1):67–82, Jan. 2001.

J.D. Collins and D.M. Tullsen. Runtime identification of cache conflict misses: The adaptive miss buffer. *ACM Transactions on Computer Systems (TOCS)*, 19(4):413–439, Nov. 2001.

J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In *Int'l Conf. on Supercomputing (ICS)*, 2003.

F. J. Corbato. A paging experiment with the Multics system. MAC Report MAC-M-384, Massachusetts Institute of Technology, May 1968.

P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, Oct. 1971.

R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. of Research & Development*, 25(5):483–490, Sept. 1981.

J.M. Denham, P. Long, and J.A. Woodward. DEC OSF/1 version 3.0 symmetric multiprocessing implementation. *Digital Technical J. of Digital Equipment Corporation*, 6(3):29–43, Summer 1994.

P.J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5): 323–333, May 1968.

P.J. Denning. Performance modeling: Experimental computer science at its best. *Communications of the ACM*, 24(11):725–727, Nov. 1981.

P.J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. AMD, Nov. 16, 2007.

M. Durbhakula. Sharing-aware OS scheduling algorithms for multi-socket multi-core servers. In *Int'l Forum on Next-Generation Multicore/Manycore Technologies (IFMT)*, 2008.

H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Int'l Conf. on High Performance Computing (HiPC)*, pages 22–34, 2006.

H. Dybdahl, P. Stenström, and L. Natvig. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 2–12, 2007.

J. Edler and M.D. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

A. El-Moursy, R. Garg, D.H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Int'l Parallel & Distributed Processing Symp. (IPDPS)*, 2006.

D.R. Engler, M.F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Symp. on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

N. Enright Jerger, D. Vantrease, and M. Lipasti. An evaluation of server consolidation workloads for multi-core designs. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 47–56, 2007.

P.H. Enslow, Jr. Multiprocessor organization–a survey. *ACM Computing Surveys*, 9(1):103–129, Mar. 1977.

S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Ottawa Linux Symp.*, pages 269–288, 2006.

S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 91–102, 2010.

F. Faggin, M.E. Hoff, Jr, S. Mazor, and M. Shima. The history of the 4004. *IEEE Micro*, 16(6): 10–20, Dec. 1996.

A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *ACM SIGOPS European Workshop*, 2004.

A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conf.*, 2005.

A. Fedorova, M. Seltzer, and M.D. Smith. A non-work-conserving operating system scheduler for SMT processors. In *Workshop on the Interaction between Operating Systems & Computer Architecture (WIOSCA)*, 2006.

A. Fedorova, M. Seltzer, and M.D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 25–38, 2007.

A. Fedorova, J.C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Communications of the ACM (CACM)*, 52(12):48–57, Dec. 2009.

A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM (CACM)*, 53(2):49–57, Feb. 2010.

R. Fitzgerald and R. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems (TOCS)*, 4(2):147–177, May 1986.

Fujitsu. Fujitsu and Sun Microsystems set the standard for open systems computing with fastest, most reliable Solaris/SPARC servers: Co-developed "SPARC Enterprise" servers: To be co-marketed by Fujitsu and Sun. Fujitsu Press Release, Tokyo, Japan, and Santa Clara, CA, Apr. 17, 2007.

Fujitsu. Fujitsu and Sun unveil new entry-level server powered by the SPARC64 VII processor and the Solaris OS: New SPARC Enterprise M3000 server delivers enterprise performance and mission-critical RAS in ultra-dense footprint. Fujitsu Press Release, Tokyo, Japan, and Santa Clara, CA, Oct. 28, 2008.

B. Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System*. PhD thesis, Dept. of Computer Science, Univ. of Toronto, 1999.

B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 87–100, 1999.

G.H. Goble and M.H. Marsh. A dual processor VAX 11/780. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 291–298, 1982.

N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *Int'l Conf. on Embedded Software (EMSOFT)*, pages 245–254, 2009.

J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 297–307, 2008.

F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 228–239, 2006.

A. Gupta, A. Tucker, and S. Urushibara. The impact of operating systems scheduling policies and synchronization methods on the performance of parallel applications. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 120–132, 1991.

M.S. Al Hakeem, J. Richling, G. Mühl, and H.-U. Heiß. An adaptive scheduling policy for staged applications. In *Int'l Conf. on Internet & Web Applications & Services (ICIW)*, pages 183–192, 2009.

L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, Mar.–Apr. 2000.

N. Hardavellas, M. Ferdman, A. Ailamaki, and B. Falsafi. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, 2009.

S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Int'l Conf. on Very Large Data Bases (VLDB)*, pages 660–671, 2004.

G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, Jul. 1998.

J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 4th edition, 2007.

R.A. Ho. *Memory page placement based on predicted cache behaviour in cc-NUMA multiprocessors*. PhD thesis, Dept. of Electrical & Computer Engineering, Univ. of Toronto, 2004.

S. Hofmeyr, C. Iancu, and F. Blagojević. Load balancing on speed. In *Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 147–158, 2010.

HP. HP strengthens industry-standard offerings with new servers and storage. HP Press Release, Palo Alto, CA, Feb. 9, 2004.

L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. In *Workshop on Design, Architecture & Simulation of Chip Multi-Processors (dasCMP)*, pages 24–33, 2005.

L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 13–22, 2006.

J. Huh, S.W. Keckler, and D. Burger. Exploring the design space of future CMPs. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 199–210, 2001.

J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Int'l Conf. on Supercomputing (ICS)*, pages 31–40, 2005.

IBM. *IBM PowerPC 970FX RISC Microprocessor User's Manual*, 1.6 edition, Dec. 19, 2005a.

IBM. IBM makes first Cell computer generally available: Cell-based BladeCenter system poised to boost performance for new applications in aerospace, oil & gas and medical industries. IBM Press Release, Armonk, NY, Sept. 12, 2006.

IBM. IBM launches world's most powerful server: "Regatta" transforms the economics of UNIX servers at half the price of competition. IBM Press Release, Armonk, NY, Oct. 4, 2001.

IBM. IBM unleashes eServer i5 systems with POWER5: IBM eServer i5 offers new economic model for small and medium sized businesses; with embedded virtualization engine technology it runs multiple operating systems. IBM Press Release, Armonk, NY, May 3, 2004.

IBM. IBM unleashes world's fastest chip in powerful new computer: Processor doubles speed without adding to energy 'footprint,' enabling customers to reduce electricity consumption by almost half; enough bandwidth to download entire iTunes catalog in 60 seconds. IBM Press Release, London, UK, May 21, 2007.

IBM. IBM unveils new POWER7 systems to manage increasingly data-intensive services: Unprecedented scale for emerging industry business models, from smart electrical grids to real-time analytics. IBM Press Release, Armonk, NY, Feb. 8, 2010.

IBM. Momentum for Power architecture technology accelerates in Japan. IBM Press Release, Tokyo, Japan, Jul. 7, 2005b.

IEEE. IEEE standard for floating-point arithmetic. Technical Report IEEE Std 754-2008, Aug. 29, 2008.

Intel. *Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Series Datasheet*, 313278-008 edition, Mar. 2008a.

Intel. *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series Datasheet, Volume 1*, 320834-003 edition, Oct. 2009.

Intel. Innovation from the start. Poster 0305/TSM/LAI/HP/5K 306766-001US, 2005a.

Intel. Intel Itanium 9300 processor raises bar for scalable, resilient mission-critical computing. Intel Press Release, San Francisco, CA, Feb. 8, 2010a.

Intel. New Intel Xeon processor pushes mission critical into the mainstream. Intel Press Release, Santa Clara, CA, Mar. 30, 2010b.

Intel. New Intel high-end Xeon server processors raise performance bar. Intel Press Release, Santa Clara, CA, Sept. 15, 2008b.

Intel. Intel ignites quad-core era: World's best microprocessor gets even better. Intel Press Release, Santa Clara, CA, Nov. 14, 2006a.

Intel. New dual-core Intel Itanium 2 processor doubles performance, reduces power consumption: Aggressive growth in Itanium hardware and software solutions deliver mission critical computing freedom. Intel Press Release, Santa Clara, CA, Jul. 18, 2006b.

Intel. Dual core era begins, PC makers start selling Intel-based PCs. Intel Press Release, Santa Clara, CA, Apr. 18, 2005b.

R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Int'l Conf. on Supercomputing (ICS)*, pages 257–266, 2004.

R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 25–36, 2007.

A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31 (3):264–323, Sept. 1999.

A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr, and J. Emer. Adaptive insertion policies for managing shared caches. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 208–219, 2008.

M.D. Janssens, J.K. Annot, and A.J. Van De Goor. Adapting UNIX for a multiprocessor environment. *Communications of the ACM*, 29(9):895–901, Sept. 1986.

S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 31–42, 2002.

S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conf.*, 2005.

Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Int'l Conf. on High-Performance Embedded Architectures & Compilers (HiPEAC)*, pages 201–215, 2010a.

Y. Jiang, E. Zhang, K. Tian, and X. Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Int'l Conf. on Compiler Construction (CC)*, 2010b.

X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li. A simple cache partitioning approach in a virtualized environment. In *Int'l Symp. on Parallel & Distributed Processing with Applications (ISPA)*, pages 519–524, 2009.

M. Kandemir, S.P. Muralidhara, S.H.K. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 505–516, 2009.

H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From chaos to QoS: Case studies in CMP resource management. In *Workshop on Design, Architecture & Simulation of Chip Multi-Processors (dasCMP)*, pages 21–30, 2006.

S.F. Kaplan, L.A. McGeoch, and M.F. Cole. Adaptive caching for demand paging. In *Int'l Symp. on Memory Management (ISMM)*, pages 114–126, 2002.

R.E. Kessler and M.D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, Nov. 1992.

J.M. Kim, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 119–134, 2000.

S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 111–122, 2004.

Y.H. Kim, M.D. Hill, and D.A. Wood. Implementing stack simulation for highly-associative memories. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 212–213, 1991.

S. Kleiman, J. Voll, J. Eykholt, A. Shivalingah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multprocessing in Solaris 2.0. In *IEEE Computer Society Int'l Conf. (COMPCON)*, pages 181–186, Spring 1992.

R. Knauerhase, P. Brett, B. Hohlt, L. Tong, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May–Jun. 2008.

P. Koka and M.H. Lipasti. Opportunities for cache friendly process scheduling. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)*, 2005.

I. Kotera. *Performance and Power Aware Cache Memory Architectures*. PhD thesis, Dept. of Computer & Mathematical Sciences, Tohoku Univ., 2009.

D. Koufaty and D.T. Marr. Hyperthreading technology in the Netburst microarchitecture. *IEEE Micro*, 23(2):56–65, Mar.–Apr. 2003.

R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Int'l Symp. on Computer Architecture (ISCA)*, 2004.

R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 408–419, 2005.

V. Kumar and J. Delgrande. Optimal multicore scheduling: An application of ASP techniques. *Lecture Notes in Computer Science: Logic Programming & Nonmonotonic Reasoning*, 5753:604–609, 2009.

R.P. LaRowe, Jr and C.S. Ellis. Page placement policies for NUMA multiprocessors. *J. of Parallel & Distributed Computing*, 11(2):112–129, Feb. 1991.

R.P. LaRowe, Jr, J.T. Wilkes, and C.S. Ellis. Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In *Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 122–132, 1991.

J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *USENIX Annual Technical Conf.*, pages 103–114, 2002.

R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Int'l Conf. on Very Large Data Bases (VLDB)*, pages 373–384, 2009.

T. Li, D. Baumberger, D.A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Conf. on Supercomputing (SC)*, 2007.

J. Liedtke. On micro-kernel construction. In *Symp. on Operating Systems Principles (SOSP)*, pages 237–250, 1995.

J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Real-Time Technology & Applications Symp. (RTAS)*, pages 213–227, 1997.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2008.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Enabling software multicore cache management with lightweight hardware support. In *Conf. on Supercomputing (SC)*, 2009.

C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 176–185, 2004.

F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2010.

R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

W. Liu and D. Yeung. Using aggressor thread information to improve shared cache management for CMPs. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 372–383, 2009.

J.L. Lo. *Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors*. PhD thesis, Dept. of Computer Science & Engineering, Univ. of Washington, 1998.

J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems (TOCS)*, 15(3):322–354, Aug. 1997.

J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. *J. of Instruction-Level Parallelism*, 6:1–24, 2004.

Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 246–257, 2009.

C.-K. Luk and T.C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers (TC)*, 48(2):134–141, Feb. 1999.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conf. on Programming Language Design & Implementation (PLDI)*, pages 190–200, 2005.

W.L. Lynch, B.K. Bray, and M.J. Flynn. The effect of page allocation on caches. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 222–225, 1992.

P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Haallberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35 (2):50–58, Feb. 2002.

J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 90–99, 2006.

J. Marathe, F. Mueller, and B. de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *Int'l Conf. on Supercomputing (ICS)*, 2005.

D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M. Upton. Hyper-Threading Technology architecture and microarchitecture. *Intel Technology J.: Hyper-Threading Technology*, 6(1):4–15, Feb. 14, 2002.

R.L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems J.*, 9(2):78–117, 1970.

L.K. McDowell, S. Eggers, and S.D. Gribble. Improving server software support for simultaneous multithreaded processors. In *Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 37–48, 2003.

R.L. McGregor, C.D. Antonopoulos, and D.S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Intl. Parallel & Distributed Processing Symp. (IPDPS)*, 2005.

M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, Aug. 1984.

J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, Feb. 1991.

K. Meng, R. Joseph, R. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 177–186, 2008.

M.R. Meswani and P.J. Teller. Evaluating the performance impact of hardware thread priorities in simultaneous multithreaded processors using SPEC CPU2000. In *Workshop on Operating System Interference in High Performance Applications (OSIHPA)*, 2006.

R.A. Meyer and L.H. Seawright. A virtual machine time-sharing system. *IBM Systems J.*, 9(3): 199–218, Sept. 1970.

Microsoft. The countdown begins: Xbox 360 to hit store shelves Nov. 22 in North America, Dec. 2 in Europe and Dec. 10 in Japan. Microsoft Press Release, Tokyo, Japan, Sept. 15, 2005.

Microsoft. Windows support for Hyper-Threading technology. *Windows Platform Design Notes*, 2002.

G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965.

G.E. Moore. Progress in digital integrated electronics. In *Int'l Electron Devices Meeting*, volume 21, pages 11–13, 1975.

Motorola. *Product Brief: MPC821 PowerPC Personal Systems Microprocessor*, MPC821/D REV 4 edition, 1998.

F. Mueller. Compiler support for software-based cache partitioning. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS)*, pages 125–133, 1995.

B. Mukherjee and K. Schwan. Experiments with a configurable lock for multiprocessors. In *Int'l Conf. on Parallel Processing (ICPP)*, pages 205–208, 1993.

B. Mukherjee, K. Schwan, and P. Gopinath. A survey of multiprocessor operating system kernels (DRAFT). Technical Report GIT-CC-92/05, College of Computing, Georgia Institute of Technology, Nov. 1993.

J. Nakajima and V. Pallipadi. Enhancements for Hyper-Threading technology in the operating system – seeking the optimal micro-architectural scheduling. In *Workshop on Industrial Experiences With Systems Software (WIESS)*, pages 25–38, 2002.

J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 89–104, 2002.

L.M. Ni and C.-F.E. Wu. Design tradeoffs for process scheduling in shared memory multiprocessor systems. *IEEE Transactions on Software Engineering*, 15(3):327–334, Mar. 1989.

K. Nikas, M. Horsnell, and J. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *Int'l Symp. on Systems, Architecture, Modeling and Simulation (SAMOS)*, pages 25–32, 2008.

D.S. Nikolopoulos. Dynamic tiling for effective use of shared caches on multithreaded processors. *Int'l J. of High Performance Computing & Networking*, 2(1):22–35, Feb. 2004.

M. Olszewski, K. Mierle, A. Czajkowski, and A. Demke Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. In *European Conf. on Computer Systems (EuroSys)*, pages 3–16, 2007.

K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 2–11, 1996.

J.K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Technical Conf.*, pages 247–256, 1990.

S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. Technical report, Dept. of Computer Science & Engineering, Univ. of Washington, 2000.

R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Symp. on Operating Systems Principles (SOSP)*, pages 79–95, 1995.

J.K. Peacock, S. Saxena, D. Thomas, F. Yang, and W. Yu. Experiences from multithreading System V Release 4. In *Symp. on Experiences with Distributed & Multiprocessor Systems (SEDMS)*, pages 77–91, 1992.

J. Philbin, J. Edler, O.J. Anshus, C.C. Douglas, and K. Li. Thread scheduling for cache locality. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 60–71, 1996.

M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 423–432, 2006.

M.K. Qureshi, D.N. Lynch, O. Mutlu, and Y.N. Patt. A case for MLP-aware cache replacement. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 167–178, 2006.

N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 2–12, 2006.

J.K. Rai, A. Negi, R. Wankar, and K.D. Nayak. On prediction accuracy of machine learning algorithms for characterizing shared L2 cache behavior of programs on multicore processors. In *Int'l Conf. on Computational Intelligence, Communication Systems & Networks (CICSyN)*, pages 213–219, 2009.

H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Workshop on Cloud Computing Security*, pages 77–84, 2009.

M. Rajagopalan, B.T. Lewis, and T.A. Anderson. Thread scheduling for multi-core platforms. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Real-Time Systems Symp. (RTSS)*, pages 298–308, 1997.

R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers (TC)*, 47(8):896–908, Aug. 1988.

J.A. Redstone. *An Analysis of Software Interface Issues for SMT Processors*. PhD thesis, Dept. of Computer Science & Engineering, Univ. of Washington, 2002.

J.A. Redstone, S.J. Eggers, and H.M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 245–256, 2000.

B.M. Rogers, A. Krishna, G.B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 371–382, 2009.

T.H. Romer, W.H. Ohlrich, A.R. Karlin, and B.N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 176–187, 1995.

M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Symp. on Operating Systems Principles (SOSP)*, pages 285–298, 1995.

RUBiS. RUBiS web page. http://rubis.ow2.org/.

C.H. Russell and P.J. Waterman. Variations on UNIX for parallel-processing computers. *Communications of the ACM*, 30(12):1048–1055, Dec. 1987.

J.C. Sáez, J.I. Gomez, and M. Prieto. Improving priority enforcement via non-work-conserving scheduling. In *Int'l Conf. on Parallel Processing (ICPP)*, pages 99–106, 2008.

B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R.L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *European Conf. on Computer Systems (EuroSys)*, pages 73–86, 2007.

F.T. Schneider. *Online Optimization Using Hardware Performance Monitor*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2009.

A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Workshop on Managed Many-Core Systems (MMCS)*, 2008.

A. Settle, J. Kihm, A. Janiszewski, and D.A. Connors. Architectural support for enhanced SMT job scheduling. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 63–73, 2004.

J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Tech. Conf.*, pages 17–30, 2005.

SGI. New dual-core SGI Altix systems push leading applications to unprecedented heights. Silicon Graphics Press Release, Jul. 18, 2006.

D. Shelepov, J.C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar. HASS: a scheduler for heterogeneous multicore systems. *ACM Special Interest Group in Operating Systems (SIGOPS), Operating Systems Review (OSR)*, 43(2):66–75, Apr. 2009.

K. Shen. Request behavior variations. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2010.

X. Shen and Y. Jiang. Co-run locality prediction for proactive shared-cache management. Technical Report WM-CS-2009-03, College of William & Mary, Mar. 24, 2009.

X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Symp. on Principles of Programming Languages (POPL)*, pages 55–61, 2007.

T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Int'l Conf. on Supercomputing (ICS)*, pages 155–164, 1999.

T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 336–349, 2003.

B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner. POWER5 system microarchitecture. *IBM J. of Research & Development: POWER5 and packaging*, 49(4/5):505–521, Jul. 2005.

J.E. Smith and R. Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, May 2005.

A. Snavely and D.M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 234–244, 2000.

A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 66–76, 2002.

L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 258–269, 2008.

G.S. Sohi. Single-chip multiprocessors: The rebirth of parallel processing. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, page 2, 2003.

T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *ICSE Workshop on Multicore Software Engineering (IWMSE)*, pages 73–80, 2009.

Sony. Playstation 3 launches on November 17, 2006 in North America. Sony Press Release, Los Angeles, CA, Nov. 17, 2006.

G. Soundararajan, J. Chen, M. Sharaf, and C. Amza. Dynamic partitioning of the cache hierarchy in shared data centers. In *Int'l Conf. on Very Large Data Bases (VLDB)*, pages 635–646, 2008.

SPEC. SPEC JBB2000 web page. http://www.spec.org/jbb2000/.

B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul.–Aug. 2002.

M.S. Squillante and E.D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel & Distributed Systems*, 4(2):131–143, Feb. 1993.

S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *Workshop on Operating System Interference in High Performance Applications (OSIHPA)*, 2006.

S. Srikantaiah and M. Kandemir. SRP: Symbiotic resource partitioning of the memory hierarchy in CMPs. In *Int'l Conf. on High-Performance Embedded Architectures & Compilers (HiPEAC)*, pages 277–291, 2010.

S. Srikantaiah, M. Kandemir, and M.J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 135–144, 2008.

S. Srikantaiah, R. Das, A.K. Mishra, C.R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Conf. on Supercomputing (SC)*, 2009a.

S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: Controlled shared cache management in chip multiprocessors. In *Int'l Symp. on Microarchitecture (MICRO)*, pages 517–528, 2009b.

A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Conf. on Programming Language Design & Implementation (PLDI)*, pages 196–205, 1994.

H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers (TC)*, 41(9):1054–1068, Sept. 1992.

C.D. Sudheer, T. Nagaraju, P.K. Baruah, and A. Srinivasan. Optimizing assignment of threads to SPEs on the Cell BE processor. Technical Report TR-080906, Florida State Univ., 2008.

E.G. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 116–132, 2001a.

E.G. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Int'l Conf. on Parallel & Distributed Computing Systems (PDCS)*, pages 429–443, 2001b.

E.G. Suh, L. Rudolph, and S. Devadas. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 117–128, 2002.

E.G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The J. of Supercomputing*, 28(1):7–26, Apr. 2004.

M.A. Suleman, O. Mutlu, M.K. Qureshi, and Y.N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 253–264, 2009.

Sun. Sun Microsystems announces the first implementation of the MAJC architecture – the dual processor MAJC 5200. Sun Press Release, Santa Clara, CA and Hanover, Germany, Oct. 5, 1999.

Sun. Sun brings personal visualization to the desktop; introduces high performance XVR-1000 3D graphics accelerator; advanced graphics and industry's first 24-inch flat panel monitor featuring digital interface delivers high-end visualization capabilities to the desktop. Sun Press Release, San Jose, CA, Mar. 14, 2002.

Sun. Sun Microsystems marks new era in network computing with breakthrough CoolThreads technology - unveils high-performance, eco-responsible server line. Sun Press Release, New York, NY, Dec. 6, 2005.

Sun. Sun Microsystems and Fujitsu expand SPARC enterprise server line with first systems based on the UltraSPARC T2 processor. Sun Press Release, Las Vegas, NV, Oct. 7, 2007.

Sun. Sun's new systems, software and solutions protect customer investments while pumping up performance and security. Sun Press Release, Feb. 10, 2004a.

Sun. Sun unveils industry's first throughput computing systems and complete refresh of industry's leading "Unix" systems lineup. Sun Press Release, Feb. 10, 2004b.

D. Tam. Performance analysis and optimization of the Hurricane file system. Master's thesis, Dept. of Electrical & Computer Engineering, Univ. of Toronto, 2003.

D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems & Computer Architecture (WIOSCA)*, pages 26–33, 2007a.

D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *European Conf. on Computer Systems (EuroSys)*, pages 47–58, 2007b.

D.K. Tam, R. Azimi, L.B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 121–132, 2009.

J.M. Tendler, J.S. Dodson, J.S. Fields, Jr, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM J. of Research & Development: IBM POWER4 System*, 46(1):5–26, Jan. 2002.

Texas Instruments. *TMS320C80 Digital Signal Processor Data Sheet*, SPRS023B REV Oct. 1997 edition, Jul. 1994.

R. Thekkath and S.J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 176–186, 1994.

D. Thiebaut, H. Stone, and J. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers (TC)*, 41(6):665–676, Jun. 1992.

R.H. Thompson and J.A. Wilkinson. The D825 automatic operating and scheduling program. In *AFIPS Spring Joint Computer Conference (SJCC)*, pages 41–49, 1963.

M.M. Tikir and J.K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Conf. on Supercomputing (SC)*, pages 46–57, 2004.

J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 162–174, 1992.

J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 272–274, 1993.

J. Torrellas, C. Xia, and R.L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions On Computers*, 47(12):1363–1381, Dec. 1998.

D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 392–403, 1995.

R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: a structure for scalable multiprocessor operating system design. *The J. of Supercomputing*, 9(1-2):105–134, Mar. 1995.

S. Vaddagiri, B.B. Rao, V. Srinivasan, A.P. Janakiraman, B. Singh, and V.K. Sukthankar. Scaling software on multi-core through co-scheduling of related tasks. In *Linux Symp.*, pages 287–295, 2009.

M. Varian. VM and the VM community: Past, present, and future. In *SHARE*, 1989.

R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Symp. on Operating Systems Principles (SOSP)*, pages 26–40, 1991.

T. Venton, M. Miller, R. Kalla, and A. Blanchard. A Linux-based tool for hardware bring up, Linux development, and manufacturing. *IBM Systems J.*, 44(2):319–329, Jan. 2005.

B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Conf. on Programming Language Design & Implementation (PLDI)*, pages 279–289, 1996.

Volano. Volano web page. http://www.volano.com/.

D.W. Wall. Limits of instruction-level parallelism. Research Report WRL-93-6, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, CA, Nov. 1993.

T. Walsh. Generating miss rate curves with low overhead using existing hardware. Master's thesis, Dept. of Computer Science, Univ. of Toronto, 2009.

B. Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 127–138, 1998.

E.W. Weisstein. Composition. From MathWorld – A Wolfram Web Resource http://mathworld.wolfram.com/Composition.html.

M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symp. on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM Special Interest Group in Operating Systems (SIGOPS), Operating Systems Review (OSR)*, 43(2):76–85, Apr. 2009.

A. Whitaker, M. Shaw, and S.D. Gribble. Scale and performance in the Denali isolation kernel. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 195–209, 2002.

K.M. Wilson and B.B. Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *Conf. on Supercomputing (SC)*, 2001.

A. Wolfe. Software-based cache partitioning for real-time applications. In *Int'l Workshop on Responsive Computer Systems (RCS)*, 1993.

M.-J. Wu and D. Yeung. Scaling single-program performance on large-scale chip multiprocessors. Technical Report UMIACS-TR-2009-16, Univ. of Maryland, Nov. 25, 2009.

C. Xia and J. Torrellas. Comprehensive hardware and software support for operating systems to exploit MP memory hierarchies. *IEEE Transactions on Computers (TC)*, 48(5):494–505, May 1999.

F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs. In *Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 163–180, 2008.

Y. Xie and G.H. Loh. Dynamic classification of program memory behaviors in CMPs. In *Workshop on Chip Multiprocessor Memory Systems & Interconnects (CMP-MSI)*, 2008.

Y. Xie and G.H. Loh. Scalable shared-cache management by containing thrashing workloads. In *Int'l Conf. on High-Performance Embedded Architectures & Compilers (HiPEAC)*, 2010.

C. Xu, X. Chen, R.P. Dick, and Z.M. Mao. Cache contention and application performance prediction for multi-core systems. In *Int'l Symp. on Performance Analysis of Systems & Software (ISPASS)*, 2010.

T. Yang, E.D. Berger, S.F. Kaplan, and J.E.B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *Symp. on Operating Systems Design & Implementation (OSDI)*, pages 103–116, 2006.

T. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *Int'l Conf. on Compilers, Architecture & Synthesis for Embedded Systems (CASES)*, pages 237–248, 2005.

S. Youn, H. Kim, and J. Seoul. A reusability-aware cache memory sharing technique for high performance CMPs with private L2 caches. In *Workshop on Chip Multiprocessor Memory Systems & Interconnects (CMP-MSI)*, 2007.

E.Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 203–212, 2010.

J. Zhang, X.-Y. Fan, and S.-H. Liu. A pollution alleviative L2 cache replacement policy for chip multiprocessor architecture. In *Int'l Conf. on Networking, Architecture, & Storage (NAS)*, pages 310–316, 2008.

M. Zhang and K. Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 336–345, 2005.

X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European Conf. on Computer Systems (EuroSys)*, pages 89–102, 2009a.

X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conf.*, 2009b.

L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 339–349, 2007a.

L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell. Performance, area and bandwidth implications on large-scale CMP cache design. In *Workshop on Chip Multiprocessor Memory Systems & Interconnects (CMP-MSI)*, 2007b.

Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *Int'l Symp. on Code Generation & Optimization (CGO)*, pages 299–311, 2007c.

P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 177–188, 2004.

S. Zhou and T. Brecht. Processor-pool-based scheduling for large-scale NUMA multiprocessors. In *Int'l Conf. on the Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 133–142, 1991.

X. Zhou, W. Chen, and W. Zheng. Cache sharing management for performance fairness in chip multiprocessors. In *Int'l Conf. on Parallel Architectures & Compilation Techniques (PACT)*, pages 384–393, 2009.

Y. Zhou, J.F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conf.*, pages 91–104, 2001.

S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Int'l Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2010.

C. Zilles and G. Sohi. A programmable co-processor for profiling. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2001.