# Operating System Scheduling for Chip Multithreaded Processors

A thesis presented

by

Alexandra Fedorova

to

The Division Of Engineering And Applied Sciences
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts
September, 2006

Advisor: Margo I. Seltzer Author: Alexandra Fedorova

Operating System Scheduling for Chip Multithreaded Processors

## Abstract

This dissertation addresses operating system thread scheduling for chip multithreaded processors. Chip multithreaded processors are becoming mainstream thanks to their superior performance and power characteristics. Threads running concurrently on a chip multithreaded processor share the processor's resources. Resource contention, and accordingly performance, depends on characteristics of the co-scheduled threads. The operating system controls thread co-scheduling, and thus affects performance of a chip multithreaded system.

This dissertation describes the design and implementation of three new scheduling algorithms for chip multithreaded processors: the *non-work-conserving* algorithm, the *target-miss-rate* algorithm, and the *cache-fair* algorithm. These algorithms target contention for the second-level cache, a recognized performance-critical resource, and pursue several objectives: performance optimization, fairness, and performance predictability. These algorithms use novel analytical performance models and online performance monitoring, and do not require input from applications or changes to existing hardware structures. This dissertation describes the implementation of these algorithms in a commercial operating system and evaluates their effectiveness.

*Dedicated to the memory of my grandfather*

*Nikolai Andreevich Fedorov for his relentless*

*energy, hard work and selflessness*

## ACKNOWLEDGEMENTS

While I appear as the only author of this dissertation, many people contributed to this work.

My advisor Margo Seltzer has been a mentor, a role model and a friend. Margo took me on board when I changed research areas and helped me become an inspired scientist. She has always had unwavering confidence in my success, inspiring me to accomplish milestones that had seemed out of touch. Margo let me freely explore research directions even if they did not perfectly align with her own interests, providing just enough guidance to keep me focused and motivated, while letting me make my own mistakes. Margo's advising style helped me acquire experience and maturity. Margo is one of the most fascinating people I have ever met. If doing a Ph.D. with Margo has made my personality at least a bit like hers, it was definitely worth it.

Chris Small was a key person to promote my career. Chris invited me to work at Sun Microsystems as his intern and led me to discover my dissertation topic. Chris' resourceful mind invented a concept of "permanent summer intern", which allowed me to be a Sun employee for over three years while working on my dissertation. This has had a tremendous impact on my professional growth. My dissertation and career would not have been the same without Chris.

I consider Mike Smith my second advisor. When I asked Mike to be on my committee he said he did not want to be "just another happy reader". He wanted to be involved in the work. And he has been! Mike's insight as an architect was critical to my research, which spans the areas of operating systems and computer architecture. Mike is one of the nicest people I have ever met, and one of the smartest. He would always spot subtle errors in my algorithms, and many times his ability to look ten steps ahead saved me from trying out futile solutions.

Graduate school is notorious for its stresses. And I am extremely grateful to my husband Anton for sticking with me throughout this process, supporting and encouraging me, being interested and respectful of my research, setting an example of great work ethics, and tolerating my grumpiness during paper deadlines.

I am greatly indebted to many people at Sun Microsystems Labs. The labs management (Miriam Kadansky, Steve Heller, Bob Sproull) made it possible to continue my internship for over three years despite tight budget constraints. I had an honor of working with Mark Moir, enjoying his excellent leadership skills, great work ethics, and dedication. Dan Nussbaum is one of the best programmers I know, and I owe it to him for improving my development skills and raising my standards for writing code. I thank Jim Waldo for help with my career, and for constantly reminding me that "the best thesis is the one that is done"! I also thank Yossi Lev, John Howard, Victor Luchangco, Nir Shavit, Maurice Herlihy, Jan-Willem Maessen, Dave Dice, Christine Flood, Dave Chase, Gary Cutbill, Dave Detlefs, Heidi Peabody and the rest of the labs East team for always being ready to help, for coming to my talks and asking me hard questions, and for being great colleagues.

My officemate and friend Kostas Magoutis was one of the first people to show me how to run experiments and present data. I will never forget those long days of cracking our heads over data, rebooting servers after a kernel crash and re-running experiments for the hundredth time, and then, when we'd get too tired, snacking on a baguette and fois gras and practicing salsa moves, carefully balancing around the dozen server machines in our office.

I would also like to thank my friend Galya Davidyuk, for being an inspiration when she completed her dissertation on cloned bovines despite the death of her experimental cow. I thank Arkadiy Turevskiy and Mickael Graham for being my friends during the Ph.D.

Thank you, all. This was a great journey.

# TABLE OF CONTENTS

# LIST OF TABLES AND FIGURES

# Chapter I. INTRODUCTION

## I.1 Evolution of Microprocessors

Over the past four decades, the microchip industry has been governed by Moore's law. Transistor density doubled approximately every 18 months, producing bi-annual doubling of processor clock speeds (Figure 1-1) [36]. As transistors get smaller, they can switch faster between one state and another. This has permitted to continually increase processor clock frequencies at the same time as increasing transistor densities. In 2004, the trend changed; Intel's cancellation of its 4GHz line of Pentium 4 processors heralded the end of an era [34].

Building faster processors no longer paid off. Faster processors

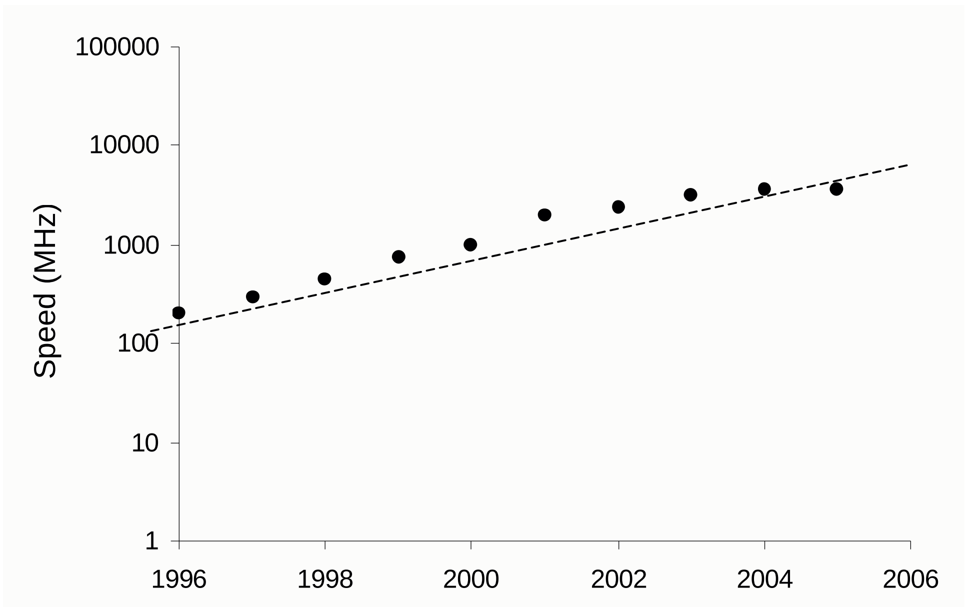


**Figure 1-1. Evolution of Intel microprocessor speed**

had higher power consumption and thermal dissipation and offered little performance improvement for modern applications. Popular modern workloads, such as databases, web servers, and application servers, are notorious for high processor cache miss rates, causing frequent memory stalls. As a result, they exhibit poor processor utilization, often less than 25%, wasting 75% of the processor's cycles handling cache misses [12]. Benchmarks from the SPEC CPU89 suite have an average processor issue slot utilization of 19%, largely due to processor cache misses [49]. If the processor is stalled on memory the majority of the time, its computational hardware is unused, and there is no benefit in making the hardware faster.

State-of-the-art architectural methods, such as pre-fetching and out-of-order execution, are used for reducing instruction-stream memory latency. However, these methods are not effective against long memory latencies; and memory latencies have reached 300 processor cycles at the time of this writing. Furthermore, the gap between processor and memory performance is growing at the rate of 50% per year [78].

These trends caused hardware manufacturers to focus on *chip multithreaded* processor technology: *chip multiprocessing* [17] and *hardware multithreading*[1] [46,49]. Chip multiprocessors, or *multicore*

[1] There are multiple terms used to describe hardware multithreading: simultaneous multithreading (SMT), interleaving, hyper-threading. We explain the difference in Chapter II.

processors, include multiple processing cores on a single chip. Hardware-multithreaded processors are equipped with multiple thread contexts. Chip multithreading enables hardware parallelism, allowing multiple application threads to run simultaneously on a single chip.

Multiple studies have demonstrated improved application throughput [17,31,48,71] and performance/watt ratio [45] of chip multithreaded architectures. This is hardly surprising, because parallelism allows for improved utilization of the processor pipeline: while one thread stalls on memory others can use the processor's functional units. Continuous improvement in transistor density results in increasing the degree of parallelism of these processors at reasonable costs. Most new processors released in 2006 by major vendors, such as AMD, Fujitsu, IBM, Intel and Sun Microsystems, are chip multithreaded [9,41,44,51,52]. According to Intel, 85% of its server processors and 70% of its PC processors will be dual-core by the end of 2006 [14].

## I.2 New Challenges for Operating System Design

Chip multithreaded processors are only now reaching the mainstream, and the community has little understanding of how to design system software that takes advantage of this hardware. This dissertation addresses operating system design for chip multithreaded processors.

Operating systems mediate access to shared hardware resources to assure optimal performance and fairness [74]. Conventional operating systems treat the CPU as a single indivisible resource, and the CPU is shared by *timeslicing*, i.e., allocating CPU time slots to threads. This makes sense for conventional single-threaded processors, because a thread gets the exclusive use of the entire CPU while it runs. On chip multithreaded processors, concurrent threads share the CPU's internal resources, such as processor caches. Therefore, treating the CPU as an indivisible resource diminishes the operating system's control over resource sharing. This dissertation demonstrates the benefits of operating system scheduling algorithms that consider internal resource sharing on chip multithreaded processors.

We focus on a specific shared resource: the second-level (L2) processor cache. This dissertation and other work [35,71] demonstrates that a shared L2 cache is a performance-critical resource on chip multithreaded processors. Sharing a performance-critical resource introduces three system performance issues[2]. We highlight these issues here and support our claims with data in Chapter II.

The first issue is *performance degradation due to cache contention* occurring when dozens of threads compete for an L2-cache (sized

[2] While the issues we highlight also apply to the L1 cache, we focus on addressing them with respect to the L2 cache, because the L2 cache has a greater impact on performance, as we demonstrate in Chapter II.

comparably to dedicated caches on single-threaded processors [3,5,41,44,52]). Effective management of the L2 cache can prevent performance degradation. The operating system controls which threads run in parallel, and, therefore, which threads share the cache. So a scheduling policy can be an effective tool for managing cache contention and system performance.

The second issue is *performance predictability*. We introduce the term *co-runner,* which refers to threads concurrently sharing a CPU core. The processor allocates the shared cache to co-runners in an ad-hoc manner, i.e., based on each co-runner's demands. Therefore, the amount of cache given to a thread depends on its co-runner. As a result, a thread's performance depends on its co-runner. This produces performance that is far less predictable than on conventional processors, posing problems for performance tuning, quality-of-service and soft real-time applications.

The third issue is *fairness*. Cache allocation on chip multiprocessors is hidden from the operating system. Most processors completely mask any information on cache allocation from the operating system. Although some chip multiprocessors, such as Intel's Core Duo [37], allow measuring the number of cache lines allocated to each co-runner, they provide no mechanism to enforce fair cache sharing. Unfair cache sharing leads to unfair CPU sharing, causing inaccurate accounting on shared multi-user systems with usage-based payment structure [2,7].

## I.3 Thesis Contributions

This dissertation addresses performance degradation, performance predictability and fairness with three new scheduling algorithms: the *non-work-conserving* algorithm, the *target-miss-rate* algorithm, and the *cache-fair* algorithm.

The **non-work-conserving algorithm** addresses performance degradation due to cache contention. It detects when L2 cache contention leads to throughput degradation and then schedules fewer concurrent threads in order to reduce contention. The algorithm uses a new online analytical model to determine the number of concurrent threads yielding the best performance.

The **target-miss-rate algorithm** uses an alternative approach to address cache contention. It schedules threads such that the processor maintains a target L2 cache miss rate, improving overall system throughput. The algorithm dynamically identifies the threads that contribute the most to the overall miss rate and then lowers the CPU share for such threads. By maintaining a target miss rate this algorithm maintains predictable system throughput. We found that while maintaining predictable throughput favors performance predictability, excessive focus on throughput may sacrifice fairness.

Our final algorithm, the **cache-fair algorithm**, increases performance predictability, while actually improving fairness and overall system throughput. The algorithm works by reducing effects of unequal CPU cache sharing on threads' performance. The algorithm dynamically determines each thread's cache miss rate achieved under fair cache sharing, and if the thread's miss rate deviates from its fair miss rate, the algorithm compensates by adjusting the thread's CPU time quantum. As a result, applications achieve performance close to what they would achieve under fair cache sharing.

The work presented in this dissertation explores new and unique approaches in chip-multithreading-aware scheduling. Our algorithms are implemented purely in software and do no rely on *co-scheduling* (i.e., matching threads with co-runners so as to produce a desired effect on system performance). In contrast, existing chip-multithreading-aware algorithms use co-scheduling [39,59,68] or rely on special hardware structures [23,29,39,42,59,60,63,68,73,75].

Although our algorithms could benefit from better performance counters and control structures in chip multithreaded processors, we believe that it would be unrealistic to implement our algorithms in hardware for reasons of complexity and flexibility. Therefore, this work will continue being relevant even as chip multithreaded hardware evolves.

The contributions of this dissertation can be summarized as follows:

*Scheduling algorithms:*

– Designed the non-work-conserving scheduling algorithm, implemented a user-level prototype, and evaluated it on a simulated UltraSPARC® T1 core.

– Designed the target-miss-rate scheduling algorithm, implemented it in the Solaris™ 10 operating system, and evaluated it on a simulated UltraSPARC T1 chip multithreaded processor.

– Designed the cache-fair scheduling algorithm, implemented it in the Solaris™ 10 operating system, and evaluated it on a simulated dual-core processor based on the UltraSPARC T1 architecture.

*Performance models:*

– Designed and evaluated an online performance model to estimate the cache miss rate under fair cache sharing on a chip multiprocessor. We use this model in the cache-fair scheduling algorithm.

– Adapted an existing cache model for single-threaded processors to work for multithreaded processors. We use this model in the target-miss-rate algorithm.

– Designed an efficient online model for determining the miss rate achieved by a group of threads sharing a cache of a chip multithreaded processor. We use this model in the target-miss-rate algorithm.

– Designed an online performance model to determine the degree of concurrency yielding the maximum throughput on a multithreaded processing core. Evaluated the model for the UltraSPARC T1 architecture. We use this model in the non-work-conserving scheduler.

*Performance evaluation:*

– Studied first- and second-level caches as sources of performance bottlenecks on chip multithreaded processors, using the simulated UltraSPARC T1 architecture as the test platform. Discovered that contention for the second-level cache has a significant impact of the overall throughput, while effect of contention for the first-level cache on performance is small.
– Demonstrated that the second-level cache is not shared fairly on chip multiprocessors and that unequal cache sharing leads to co-runner dependent performance variability on chip multiprocessors.

The rest of the dissertation is structured as follows.

Chapter II provides background on chip multithreaded processor architectures and introduces the hardware platform and the workload we use for evaluation. It presents data demonstrating performance critical qualities of the L2 cache and shows how unequal cache sharing leads to poor performance predictability. This chapter concludes by highlighting the state-of-the-art on chip-multithreading-aware operating systems research.

Chapter III presents the non-work-conserving scheduling algorithm. It discusses the design of the algorithm, focusing on the new analytical performance model, and presents the evaluation using a user-level scheduler prototype.

Chapter IV describes the target-miss-rate algorithm, its implementation in Solaris™ 10, and the evaluation on a simulated UltraSPARC T1 platform. This chapter also presents a study evaluating general applicability of our results.

Chapter V presents the cache-fair scheduling algorithm and describes its implementation in Solaris 10 and evaluation on a simulated UltraSPARC T1 processor. We show that the algorithm improves CPU-sharing fairness, increases performance predictability, and boosts overall system throughput.

Chapter VI discusses related work in the areas of operating system scheduling for chip multithreaded processors and analytical performance modeling.

Chapter VII summarizes the contributions and lessons of this dissertation, provides insight on designing future chip multithreaded systems, and outlines future research.

# Chapter II. BACKGROUND

This chapter provides background on the chip multithreaded architecture, describes system performance problems arising specifically on this architecture and outlines the research addressing these problems. We begin by describing the key differences between chip multithreaded and conventional processors, focusing on shared second-level caches. We then introduce the chip multithreaded platform and the benchmarks we use for our experiments. We follow by demonstrating that the shared second-level (L2) cache is a performance critical resource. In particular, we show that insufficient L2 cache causes throughput degradation, and that unequal sharing of the L2 cache leads to co-runner dependent performance. We conclude by outlining existing approaches to addressing these problems.

## II.1 Chip Multithreaded Hardware Architecture

Chip multithreaded processors allow running multiple threads in parallel on a single chip, using a processor's *thread contexts*, often referred to as *virtual processors*. Hardware parallelism is achieved using one or both of the following methods: hardware multithreading and chip multiprocessing.

### II.1.1 Hardware Multithreading

Hardware multithreading can be broadly categorized as *coarse-grained*, *fine-grained,* and *simultaneous*. They differ in how the processor switches among the thread contexts.

A *coarse-grained multithreaded* processor switches to a new thread context when a running thread stalls on a memory request or other long-latency operation [11]. The downside of this design is a high cost of thread switching. The processor decides whether to switch a thread during a late stage in the pipeline, when the thread has already several instructions in flight. At this point, the processor has wasted its cycles on issuing instructions that would be squashed.

A solution to a high thread-switching cost came in *fine-grained multithreaded,* or *interleaved*, processor architectures, where the processor switches among the threads on every cycle [44,45].

*Simultaneous multithreading* (SMT), or *hyper-threaded* architectures add hardware multithreading to multiple-issue, out-of-order processors. They issue multiple instructions on each cycle, drawing them from multiple instruction streams, converting thread-level parallelism into instruction-level parallelism [49,51,75].

Hardware resources of a processing core are typically shared among the thread contexts. The degree of sharing depends on the architecture. First-level caches, i.e., the instruction and data cache, are typically shared. Intel's hyper-threaded Pentium 4 has a shared 8KB data cache, and a shared 12KB instruction cache [51]. Sun Microsystems' UltraSPARC T1 has a shared 8KB data cache and a shared 16KB instruction cache [44].

### II.1.2 Chip Multiprocessing

Chip multiprocessors (CMP) combine several fully-functional processing cores on a single chip [17]. This is another way to achieve hardware parallelism.

Each processing core may also be multithreaded (Figure 2-1). Chip multiprocessors with single-threaded [1,3,5,38] and multithreaded cores are available today [41,44,52].

Chip multiprocessors are increasingly built with shared second-level (L2) caches [3,41,44,52], although private-cache designs are also available [5,9,38].

Chip multiprocessors come in various configurations. IBM's

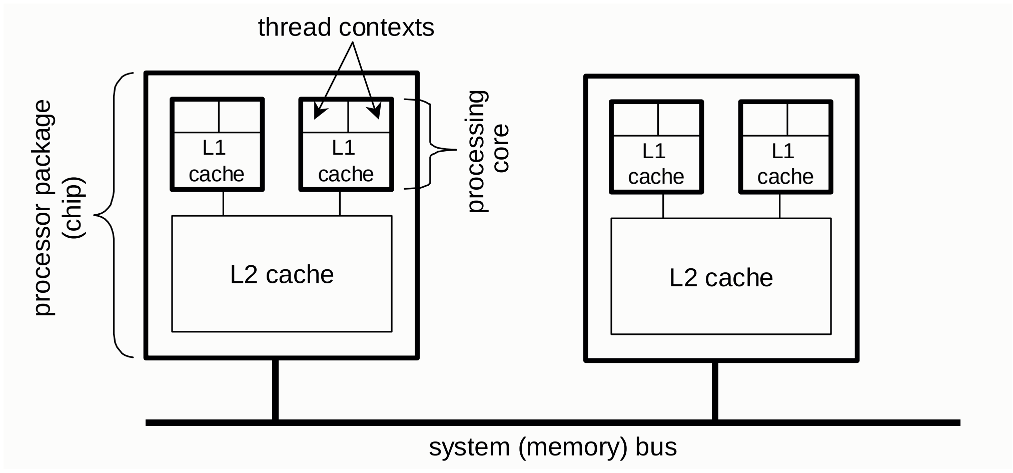


**Figure 2-1. A schematic view of a chip-multiprocessor with multithreaded cores**

Power5 chip multiprocessor [33] has two multithreaded cores, and each core has two thread contexts. There is a shared 1.875MB L2 cache on the chip. Sun Microsystems' UltraSPARC T1 [44] has eight multithreaded (interleaving-based) cores on a chip. Each core has four thread contexts, allowing the processor to run 32 threads in parallel on a single chip. The cores share a 3MB L2 cache. Fujitsu's UltraSPARC® IV processor has two two-way multithreaded cores [52]. Intel's Core Duo has two single-threaded cores and a shared 1MB L2 cache [3].

## II.2 Experimental Environment

### II.2.1 Experimental Processor

In this dissertation we present experiments performed on a simulated processor based on the UltraSPARC T1 architecture. Our hardware simulator [57] is built on top of Simics [50]. This is a *full-system* simulator: it boots the Solaris operating system and runs programs that would run on a Solaris/SPARC platform. *Therefore, the operating system runs just like it would on a real hardware, and the scheduling algorithms described in this dissertation are implemented exactly as they would be for a real machine.* All the simulations described in this paper are execution-driven including both user-level and operating system code.

We simulate the UltraSPARC T1's single-issue interleaving-based processing core. We use single-threaded and four-way multithreaded configurations in our experiments. We accurately simulate pipeline contention, the L1 cache, bandwidth limits on crossbar connections between the L1 and L2 cache, the L2 cache, and bandwidth limits on the path between the L2 cache and main memory. Each core has an 8KB L1 data cache and a 16KB L1 instruction cache (both 4-way set-associative). There is a unified set-associative L2 cache, shared among all cores on the chip. We configure the degree of associativity and the cache size depending on the experiment. We provide details of the specific hardware configuration used for each experiment in subsequent chapters.

### II.2.2 Experimental Workload

For most experiments presented in this dissertation we use a subset of benchmarks from the SPEC CPU2000 suite [6].

| | |
|---|---|
| *ammp* | – computational chemistry |
| *art* | – image recognition |
| *bzip* | – a compression utility |
| *crafty* | – game playing: chess |
| *eon* | – computer visualization |
| *gap* | – group theory, interpreter |
| *gcc* | – a compiler |
| *gzip* | – a compression utility |
| *mcf* | – combinatorial optimization |
| *parser* | – word processing |
| *twolf* | – place and route simulator |
| *vortex* | – object-oriented database |
| *vpr* | – FPGA circuit placement and routing |

SPEC CPU2000 benchmarks are comprised of real programs, represent a variety of caching behaviors, and have architectural characteristics similar to more complex benchmarks [47].

## II.3 Effects of L2 Cache Contention on Performance

The second-level (L2) cache is a performance-critical shared resource. Missing in the L2 cache requires main memory (DRAM) access, costing hundreds of processor cycles. The cost of memory access has reached roughly 300 processor cycles in 2006 and has been steadily increasing at the rate of 50% per year [78]. In contrast, missing in the L1 cache costs only tens of processor cycles. Several studies show that insufficient L2 cache has a critical effect on application performance

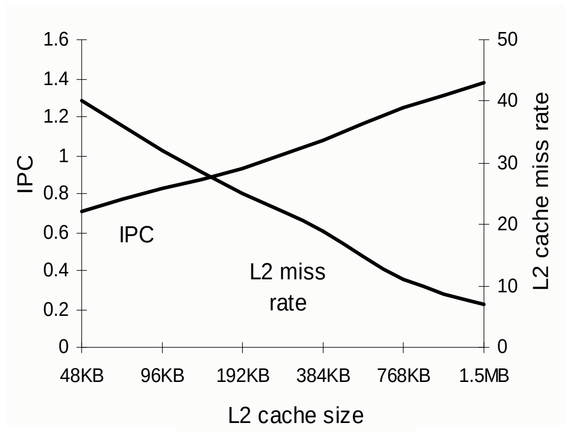


**Figure 2-2. Effect of the L2 cache miss rate on the instructions per cycle (IPC).** L2 cache miss rate in misses per 100 ref. (right-hand Y-axis) and processor IPC (left-hand Y-axis) for a multi-program SPEC CPU 2000 workload. The simulator is configured with two cores, each with four thread contexts, and a varying L2 cache size. Processor IPC significantly degrades as the L2 miss rate increases.

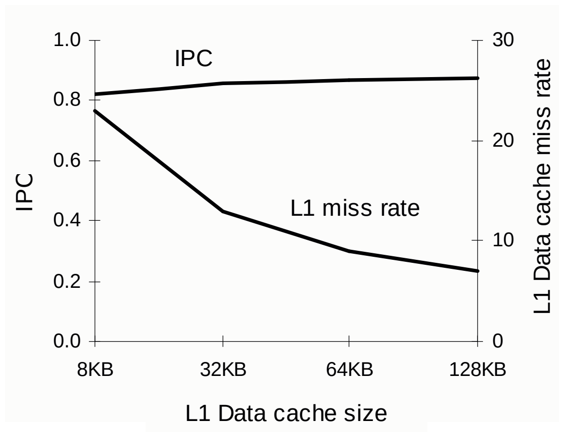


**Figure 2-3. Effect of the L1 data cache miss rate on the IPC.** L1 data cache miss rate in misses per 100 ref. (right-hand Y-axis) and processor IPC (left-hand Y-axis) for a multi-program SPEC CPU 2000 workload. The simulator is configured with a single core with four thread contexts, a varying L1 data cache size, and a 3MB L2 cache. Processor IPC is largely insensitive to the variation in the L1 data cache miss rate.

[35,71].

Consider an experiment showing the effect of L1 (data) and L2 cache miss rates on system performance. We use a workload consisting of nine SPEC CPU2000 integer benchmarks: *gzip, vpr (place), vpr (route), gcc, art, crafty, ammp, parser, vortex.* We run two copies of each of these benchmarks concurrently, producing a multithreaded workload of 18 threads.

Figures 2-2 shows the variation in the L2 cache miss rate and in the system instructions per cycle (IPC) as the L2 cache size increases. Figure 2-3 shows the variation in the L1 cache miss rate and IPC as the

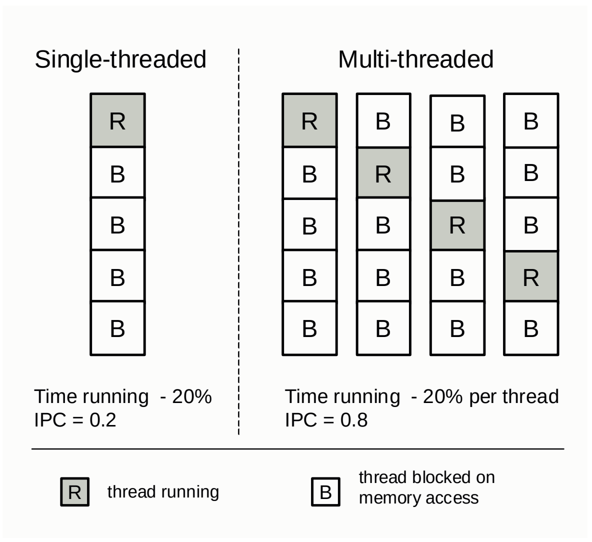


**Figure 2-4. Masking memory latency by hardware multithreading.**
Each box represents the state of the processor pipeline for a single cycle. For a single-threaded processor (left), if a thread spends 20% of its time running and the remainder of its time stalled handling cache misses, the processor stalls 80% of the time and completes only one instruction in five cycles, yielding 0.2 instructions per cycle (IPC). A multithreaded processor (right) hides memory latency. Although each thread stalls 80% of the time, overall, the processor stalls only 20% of the time, yielding 0.8 IPC.

L1 cache size increases (L2 cache is fixed at 3MB). Although the cache miss rates drop significantly with the increasing cache size in both figures, the IPC varies only in response to the variation in the L2 cache miss rate. Hardware multithreading masks short L1 cache-miss latencies (Figure 2-4), but it cannot mask long L2 cache-miss latencies.

These experiments demonstrate that managing contention for the L2 cache can have a significant payoff for performance. Although architects attempt to find the “right” cache size for a processor, the strategies they use have limitations. Determining the right cache size is accomplished by simulating workloads of interest during the design phase. Applications evolve more quickly than hardware, so the “right” cache size may turn out insufficient while the processor is still in use.

## II.4 Unequal Cache Sharing on Chip Multiprocessors

Chip multithreaded processors allocate cache among co-runners in an ad-hoc fashion. Cache allocation depends on cache demands of each co-runner. This may produce unequal and unfair cache sharing.

Table 2-1 demonstrates unequal cache sharing on a two-way chip multiprocessor. We run pairs of SPEC CPU2000 benchmarks on our simulated processor configured with two single-threaded cores and a shared 1MB L2 cache. We instrumented our simulator to count the number of cache lines allocated to each processing core, and we bind each co-runner to its own core. The simulation lasts until the first

**Table 2-1. Unequal cache sharing on chip multiprocessors.**

| Benchmark | | Number of cache lines allocated | |
|---|---|---|---|
| **CORE0** | **CORE1** | **CORE0** | **CORE1** |
| *vortex* | *mcf* | 4,023 | 12,324 |
| *vortex* | *crafty* | 7,244 | 9,095 |
| *vortex* | *gzip* | 8,712 | 7,573 |
| | | | |
| *art* | *mcf* | 7,364 | 9,000 |
| *art* | *gzip* | 13,569 | 2,782 |
| *art* | *twolf* | 7,794 | 8,585 |
| | | | |
| *crafty* | *art* | 3,853 | 12,517 |
| *crafty* | *vortex* | 9,546 | 6,737 |
| *crafty* | *gzip* | 10,745 | 5,545 |
| | | | |
| *gzip* | *mcf* | 3,379 | 12,963 |
| *gzip* | *twolf* | 3,197 | 13,143 |
| *gzip* | *vortex* | 8,003 | 8,229 |

The number of allocated cache lines is measured by sampling, so the cache lines allocated to each CPU do not always add up to 16384 – the total number of lines in the cache.
Same pairs of benchmarks (e.g., *vortex-crafty* and *crafty-vortex*) do not necessarily have same respective cache allocations. The simulation lasts until the benchmark on CORE0 completes 500 million instructions, so same pairs of benchmarks execute different sets of instructions depending on the benchmark bound to CORE0.

benchmark in the pair (bound to CORE0) completes 500 million instructions. We show the number of cache lines allocated to each core.

The processor does not allocate the cache equally among the co-runners. For example, *vortex* owns roughly more than twice as many cache lines when it runs with *gzip* than when it runs with *mcf.* We verified that the co-runner that owned fewer cache lines could use more

**Table 2-2. Co-runner dependent performance variability**

| Principal benchmark | Co-runners of the principal benchmark | | Percent slowdown in the SLOW schedule compared to the FAST schedule |
|---|---|---|---|
| | FAST schedule | SLOW schedule | |
| *twolf* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 27.58% |
| *art* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 20.31% |
| *vpr* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 18.67% |
| *gcc* | *vpr, vpr, vpr* | *mcf, mcf, mcf* | 11.88% |
| *parser* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 11.41% |
| *crafty* | *vpr, vpr, vpr* | *mcf, mcf, mcf* | 7.58% |
| *vortex* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 5.97% |
| *gzip* | *crafty, crafty, crafty* | *mcf, mcf, mcf* | 5.93% |
| *mcf* | *crafty, crafty, crafty* | *gzip, gzip, gzip* | 4.61% |

cache lines if they were available: we ran both benchmarks on a processor with an equally partitioned cache and verified that each co-runner used its entire partition.

The number of cache lines allocated to a benchmark depends on the co-runner. Co-runner dependent cache allocation leads to co-runner-dependent performance, as we demonstrate in the next section.

## II.5 Co-runner Dependent Performance Variability

Co-runner dependent cache allocation produces co-runner-dependent performance variability. Because the L2 cache is performance-critical, the variability is significant.

We use a subset of the SPEC CPU2000 benchmarks and run each benchmark, the *principal* benchmark, with two different sets of co-runners on a simulated dual-core processor with a 1MB shared L2 cache. We measure the time the principal benchmark takes to complete 500 million instructions with each set of co-runners. We do not run benchmarks to completion due to limited simulator speed. Table 2-2 summarizes the results: we show the co-runner sets that produced the shortest and longest completion times for the principal benchmark. We refer to them as "FAST" and "SLOW" schedules respectively. We also show performance variability, expressed *as percent slowdown of the principal benchmark's completion time in the SLOW schedule compared to the FAST schedule.*

All benchmarks demonstrate significant co-runner dependent performance variability: *twolf* runs 28% slower in the SLOW schedule than in the FAST schedule. For other benchmarks, the performance variability is in the range of 5-20%.

Co-runner dependent performance variability signals unfair CPU sharing, and leads to unpredictable performance and priority inversion. Pay-per-hour service providers, such as Sun Grid [7] or IBM Virtual Hosting [2], charge users per CPU hour, and unfair CPU sharing leads to incorrect accounting: Users whose applications share the processor with a "bad" co-runner do not get their money's worth. Co-runner-dependency produces less predictable performance than on conventional, single-threaded processors, complicating performance tuning and design of quality-of-service and soft real-time systems. Priority inversion occurs when a high-priority thread's does not experience a performance boost, because it runs with a "bad" co-runner. These issues present a challenge to successful adoption of chip multithreaded processors.

## II.6 Chip-Multithreading-Aware Scheduling Algorithms

This section introduces existing chip-multithreading-aware operating systems research. While Chapter VI provides a detailed comparison with the related work, this section highlights the differences between the methods used in the existing work and in the work presented in this dissertation.

Existing work on chip-multithreading-aware scheduling addressed performance, predictability and fairness via co-scheduling, in many cases relying on special hardware, such as dynamic cache partitioning and performance counters helping determine optimal cache allocations for co-runners.

Co-scheduling [40,56,59,68] refers to selecting co-runners that produce minimal contention for shared resources. Because co-runner selection algorithms may have limited scalability (the co-runners set size grows with the number of thread contexts, and the number of sets grows with the number of co-runners), we use other methods in our algorithms. The target-miss-rate algorithm uses the *CPU preference* mechanism – a co-runner-matching technique that does not require constructing co-runner sets. The cache-fair algorithm uses CPU timeslicing and does not rely on co-scheduling.

Many existing scheduling algorithms rely on special hardware support [23,29,39,42,60,63,73], such as dynamically partitioning the cache among co-runners to reduce the effects on unequal cache sharing. Our algorithms run on existing hardware, deriving information for scheduling decisions from analytical models and online measurements.

Although chip-multithreading-aware operating systems research is still nascent, this dissertation furthers the work in this area, presenting new and unique approaches to problems of performance, fairness and predictability.

# Chapter III. THE NON-WORK-CONSERVING ALGORITHM

This chapter describes a scheduling algorithm that addresses contention for the L2 cache using a *non-work-conserving scheduling* policy. Conventional operating systems schedulers are *work-conserving*: they schedule a thread on every thread context, or *virtual processor*, as long as there are runnable threads in the system. On multithreaded processors, contention for a shared cache can lead to *thrashing* [27,40] – a phenomenon where concurrent threads displace each others' working sets from the cache, experiencing extremely poor performance. In such situations, a *non-work-conserving* policy could reduce cache contention by keeping one or more thread contexts idle. The goal of this chapter is to evaluate effectiveness of a non-work-conserving scheduling policy for eliminating thrashing in the L2 cache on multithreaded processors.

## III.1 Introduction

Non-work-conserving scheduling may keep one or more virtual processors idle [61,67], even if there are runnable threads. We present a new non-work-conserving scheduling algorithm for multithreaded processors and evaluate it using a user-level scheduler prototype. At the heart of our scheduler is a new analytical model that determines when it is beneficial to leave virtual processors idle. The model estimates the workload's instructions per cycle (IPC) for a given *degree of concurrency*, i.e., the number of concurrent threads. The scheduler uses this model to determine the degree of concurrency yielding the highest IPC and then uses only that many virtual processors.

The analytical model is the key contribution of this work. Similar models [30,62] were designed for offline studies of multithreaded architectures. Their main objective was accuracy. Efficiency, simplicity, and the ability to obtain model inputs at runtime were less important.

In contrast, our model is designed specifically for use inside an operating system scheduler. The scheduler is a performance critical component of the operating system invoked hundreds of times per second, and any computation it performs must be low overhead. Therefore, our model has two critical requirements: a) model calculations must be simple enough so they can be performed in a small number of steps and implemented without floating-point operations (many modern operating systems do not permit floating-point operations inside the kernel); and b) inputs for the model must be obtainable at runtime and/or at compile time.

Our model's simplicity derives from focusing on those resources for which contention is most likely to cause thrashing. We model these resources precisely, while representing other resources with simple, less precise, models. We identify contention for the L2 cache as the prime

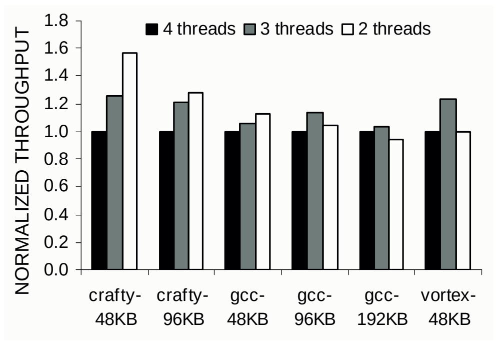


**Figure 3-1a. Normalized throughput for memory-intensive SPEC CPU2000 benchmarks, on a simulated four-way multithreaded processor**

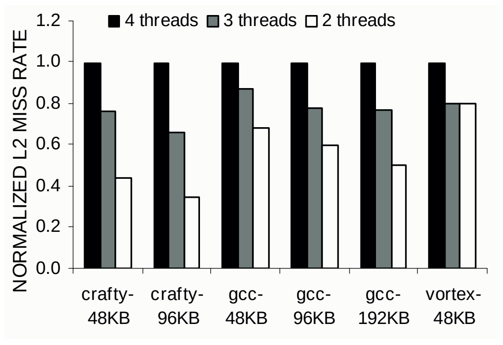


**Figure 3-1b. L2 miss rates for the experiments in Figure 3-1a**

We run multiple instances of the same benchmark concurrently, creating a multi-program workload. There is a set of bars for each benchmark/L2 cache size, and each bar corresponds to the number of concurrent benchmark instances. Using a high degree of concurrency causes thrashing (4-1a) due to contention for the L2 cache (4-1b). Lowering the degree of concurrency eliminates thrashing and improves the throughput.

cause of performance degradation and thrashing (see Section II.3 and Figures 3-1a and 3-1b). Our model precisely expresses the relationship between the L2 cache miss rate and IPC; we approximate contention for other resources using models derived via linear regression, significantly simplifying our model.

The challenge in estimating the effect of the L2 cache miss rate on the IPC of a multithreaded processor is determining how much of the L2 cache miss latency can be “hidden” by hardware multithreading: the processor overlaps memory access of one thread with computation of another. In Section III.2, we describe how we addressed this challenge. We use basic probability theory to model this effect simply. Previous models used Markov chains and were more complex [30,62].

We validate our model for the UltraSPARC T1 multithreaded processor. We use homogeneous workloads[3] (multithreaded workloads where all threads are performing similar work) constructed of the SPEC CPU2000 benchmarks. We compare our model's IPC predictions to the actual measurements obtained using an execution-driven simulator of the UltraSPARC T1 and find them to be accurate to within 10% for most workloads, with the largest observed difference between predicted and measured values of 31%. This accuracy is comparable to results for more complex off-line models [30,62].

In section III.3 we describe our user-level prototype implementation of the non-work-conserving scheduling algorithm, and demonstrate its ability to predict when it is beneficial to use the non-work-conserving policy. We show that applications using our scheduler achieve throughput improvement of 3 to 56% when the non-work-conserving policy is beneficial and perform no worse than with the default scheduler otherwise.

## III.2 The IPC Model

Our model expresses the relationship between the L2 cache miss rate and the processor's IPC. The IPC is determined by two components: 1) how many cycles the processor is busy and 2) how many cycles the processor stalls handling L2 cache misses. We refer to the first

[3] Homogeneous workloads are common in data mining and bioinformatics [4,15,24].

component as *perfect-L2-cache cycles per instruction (CPI),* or just *perfect-cache CPI.* This is the CPI we would see if all of memory fit into (and was loaded into) the L2 cache – we never had to go to main memory. The second component depends on the L2 cache miss rate. We express the IPC of a workload on a multithreaded processor as a function of its perfect-cache CPI, the L2 cache miss rate, and the number of concurrent threads:

$$IPC = F(N, L2\_MR(N), perf\_cache\_CPI(N)) \qquad \text{(III.1)},$$

where *N* is the number of concurrent threads, *perf_cache_CPI(N)* is the perfect-cache CPI, and *L2_MR(N)* is the L2 cache miss rate achieved by the *N* threads.

Our goal is to precisely model the effects of L2 cache contention on IPC. In this section, we describe this model, assuming that *perf_cache_CPI(N)* and *L2_MR(N)* are measured directly. In Section III.3, we explain how we derive the values for *perf_cache_CPI(N)* and *L2_MR(N)* at runtime using a combination of simple models and measurements obtained from hardware performance counters.

Our model relies on several architectural parameters that are statically configured for a given system:

- the number of virtual processors,
- main memory (DRAM) latency,
- L2 cache size,

– memory-bus bandwidth.

We begin by describing the performance model for a single-threaded workload. Then we extend our model for multithreaded workloads.

### III.2.1 Single-threaded model

Our single-threaded CPI model is an extension of models presented in the literature [53,69,76]. The CPI is comprised of *perf_cache_CPI* and *L2_CPI*:

$$CPI = perf\_cache\_CPI + L2\_CPI \tag{III.2}$$

*L2_CPI* is the number of cycles per instruction that a thread stalls handling L2 cache misses for a given L2 cache miss rate. *L2_CPI* depends on the L2 miss rate and the cost of each miss:

$$L2\_CPI = (L2\_RMR + L2\_WMR) * L2\_MCOST \tag{III.3}$$

– *L2_RMR* – L2 read miss rate – the number of L2 read misses per instruction, including data and instruction misses.

– *L2_WMR* – the number of L2 write misses per instruction.

– *L2_MCOST* – the cost, in cycles, of handling an L2 cache miss: this is the sum of the memory-access time and the memory-bus delay. While the memory access time is fixed, the memory-bus delay depends on contention for the bus. For now we assume that memory-bus delay is zero. We relax this assumption at the end of section III.2.

In addition to cache misses, we must account for *write-back transactions,* because they contribute to *L2_CPI.* Processors with write-back caches issue write-back transactions to write dirty cache lines back to memory. A write-back may be triggered on a cache read miss or on a write miss. We define the respective write-back rates as:

*L2_WBR_R* – read-triggered write-back rate: the number of write-backs per instruction triggered by read misses (i.e. data read misses and instruction fetches).

*L2_WBR_W* – write-triggered write-back rate: the number write-backs per instruction triggered by write misses.

To account for write-back transactions, we extend our definitions of read/write miss rates as follows:

*L2_COMB_RMR* – the L2 read miss rate including the read-triggered write-back rate:

$$L2\_COMB\_RMR = L2\_RMR + L2\_WBR\_R$$

*L2_COMB_WMR* – the L2 write miss rate including the write-triggered write-back rate:

*L2_COMB_WMR = L2_WMR + L2_WBR_W.*

While some modern multithreaded processors, such as the Pentium 4, are equipped with hardware counters to measure the write-back rate [37], we are not aware of processors that provide a way to separately measure the read-triggered and write-triggered write-backs. We found that the fraction of read-triggered and write-triggered write-backs can be accurately approximated by the fraction of each type of cache miss. For example, if 60% of the cache misses are read misses, then roughly 60% of the write-backs are read-triggered. See Appendix IX.2 for details on the write-back model.

Modern processors are equipped with *write buffers* that absorb the effect of writes: a writing thread places a store value into the buffer and proceeds without blocking. The thread stalls only when the write buffer becomes full. Therefore, write misses and the corresponding write-backs do not always stall the processor. We model the write-buffer effect using an empirically derived *write stall coefficient (WSC). WSC* expresses the fraction of write misses that stall the processor. We derive this coefficient for a given architecture. The coefficient is also application-specific: the value of the coefficient depends on the workload's write miss rate. Through experimentation with SPEC CPU2000 integer benchmarks, we found that for workloads with the combined L2 write miss rate of greater than 0.005, 90% of the writes stall the processor, while for workloads with a smaller miss rate, most of the writes are non-stalling. Accordingly, we compute the workload-specific value for the *WSC* using the following step function:

$$WSC = \begin{cases} 0.9, & L2\_COMB\_WMR > 0.005, \\ 0, & L2\_COMB\_WMR \le 0.005 \end{cases}$$

To account for the write-buffer effect in our model, we multiply the L2 write miss rate by *WSC*, thus accounting only for those write-miss cycles that stall the processor:

$$\begin{aligned} L2\_CPI = {} & (L2\_COMB\_RMR + L2\_COMB\_WMR * WSC) \\ & * L2\_MCOST \end{aligned} \qquad \text{(III.4)}$$

Using the *WSC* coefficient instead of precisely modeling the complexity of the write buffer is a reasonable simplification: for most workloads, the write buffer fully absorbs the writes, so having a detailed write buffer model is not necessary.

## III.2.2 Multithreaded model

When the processor executes a multithreaded workload, one or more virtual processors stall handling L2 cache misses while others continue executing instructions. The entire processor stalls only when all virtual processors stall. In Section III.2.1, we showed how to compute the

*L2_CPI* for single-threaded execution. In this section we show how to use the *L2_CPI* to estimate the aggregate IPC achieved by a multithreaded workload for a given L2 cache miss rate.

We first introduce a *stall probability* – the probability that an individual virtual processor stalls on an L2 cache miss. Then we combine individual stall probabilities to estimate the processor IPC.

### *III.2.2.1 Stall probability*

A stall probability captures the effect of the L2 cache miss rate on an individual virtual processor. By combining individual stall probabilities, we estimate the effect of the L2 cache miss rate on the entire processor.

If the processor is running a single thread, the probability that an individual virtual processor stalls (*prob_stall)* is trivially computed using *perf_cache_CPI* and *L2_CPI* for that thread:

$$prob\_stall = \frac{L2\_CPI}{perf\_cache\_CPI + L2\_CPI}.$$

When the processor runs multiple threads on its *V* virtual processors, the meaning of perfect-cache CPI changes, and we must adjust our formula to take this into account. The perfect-cache CPI for the multithreaded workload is the aggregate CPI achieved by *all* the threads with a perfect cache hit rate: *perf_cache_CPI(V)*. (We can estimate this quantity at runtime, as described in Section III.3.) Our formula for *prob_stall* requires the perfect-cache CPI as perceived by an *individual* virtual processor. We refer to this quantity as *perf_cache_CPI_mt_ind*, and compute it as follows:

$$perf\_cache\_CPI\_mt\_ind = perf\_cache\_CPI(V) * V \qquad \text{(III.5).}$$

Our processor switches threads on every cycle, so for each cycle that a thread is busy, it waits for the rest of the threads to take their turn using the processor. Hence we multiply *perf_cache_CPI(V)* by *V* in order to compute the individual perfect-cache CPI.

A stall probability for an individual virtual processor during a multithreaded execution (*prob_stall_mt*) is:

$$prob\_stall\_mt = \frac{L2\_CPI}{perf\_cache\_CPI\_mt\_ind + L2\_CPI} \qquad \text{(III.6).}$$

### *III.2.2.2 Modeling the multithreaded IPC*

We now combine individual stall probabilities to estimate the aggregate IPC. Let *V* be the number of virtual processors. A multithreaded processor can be in one of the following *V*+1 states:

(0) All *V* virtual processors are stalled,

(1) Exactly one virtual processor is busy, (*V-1)* are stalled,

(2) Exactly two virtual processors are busy, …

…

(V) All virtual processors are busy.

In State *V*, the processor's CPI equals to *perf_cache_CPI(V).* We say that it runs at *perf_cache_IPC(V)* – the inverse of perfect-cache CPI. In State *0*, the entire processor is stalled, so it runs at the IPC equal to zero. In the remaining states, the processor achieves an IPC less than or equal to *perf_cache_IPC(V)* – we refer to this quantity as *R_IPC(K)*, where *K* corresponds to the number of virtual processors that are *busy*. So, for example, if exactly three virtual processors are busy (*K*=*3*), the processor is running at *R_IPC(3).*

A probability that a virtual processor is busy, *prob_busy_mt,* is:

$$prob\_busy\_mt = 1 - prob\_stall\_mt$$

We compute probabilities *P(K)* that a processor is in state *K*, $(0 \leq K \leq V)$ as follows:

$$P(K) = \binom{V}{K} * prob\_busy\_mt^{K} * prob\_stall\_mt^{V-K} \qquad \text{(III.7)},$$

where *K* is the number of virtual processors that are busy in state *K*. We then compute the aggregate processor IPC for the multithreaded execution by multiplying the IPC achieved in each state by the probability of that state and summing across all states:

$$IPC = \sum_{K=0}^{V} P(K) * R\_IPC(K) \qquad \text{(III.8)}.$$

### *III.2.2.3 Deriving R_IPC*

*R_IPC(K)* is the IPC achieved on a multithreaded processor with *V* virtual processors when only *K* ($0 < K < V$) of the virtual processors are busy (i.e., only *K* threads are running concurrently). We estimate it using a linear model, whose general form is:

$$R\_IPC(K) = a * K + b * perf\_cache\_IPC(V) + c \quad \text{(III.9)},$$

where *K* is the number of busy virtual processors, *perf_cache_IPC(V)* is the perfect-cache IPC achieved by *V* threads[4], and *a, b,* and *c* are linear coefficients. We derive the coefficients for the *R_IPC* equation using linear regression applied to the data on *R_IPC* and *perf_cache_IPC* collected from simulation of SPEC CPU2000 benchmarks with a large L2 cache. A recent study suggests that this equation could be derived online using real hardware [55]: this is a more practical approach and we are interested in using it in the future. The equation we derived using the simulation data has a good fit (0.9 R-squared). The SPEC CPU2000 suite consists of benchmarks with a variety of cache access patterns [47], and we expect the resulting equation to generalize well to integer workloads.

[4] In order to use this *R_IPC* model at runtime the scheduler must obtain *perf_cache_IPC(V)*, the input to the *R_IPC* model, at runtime. We explain how the scheduler obtains this input in Section III.3.

### III.2.3 Model evaluation

We designed our model so that a non-work-conserving scheduler could improve performance by determining the degree of concurrency producing the highest IPC. The model's accuracy determines the effectiveness of performance optimizations. In this section, we evaluate our model's accuracy using the integer benchmarks in the SPEC CPU2000 suite. We derive the *WSC* coefficient and the *R_IPC* equation for our model using a subset of benchmarks, i.e., the *training set.* The training set consists of *crafty, gcc, gzip, parser, vortex* and *vpr.* We report evaluation results for the entire integer suite, except *perlbmk*: this is a multi-process benchmark, so we could not easily control the number of concurrent threads for our experiments.

For each benchmark, we use our model to estimate the aggregate IPC achieved by multiple concurrent threads, each running its own instance of this benchmark. We obtain the inputs for our model by measurement. We compare the estimated IPC with the actual IPC, which we measure by running the multiple concurrent instances of the benchmark on our simulated processor. We configure our processor with a single core, and vary the number of thread contexts and the L2 cache size. We study the sensitivity of our model to variations in two of the input factors: the L2 cache miss rate and the degree of concurrency.

For the first analysis, we run four concurrent threads and vary the L2 cache size from 48KB to 192KB. This creates variation in the L2 miss rate and consequently the IPC. Figure 3-2 shows the results. The difference between the measured and estimated IPC is 4% on average, with a median difference of 3%, and a maximum difference of 12%. The estimates were less accurate for *gzip* (12% difference from the actual IPC), because the *R_IPC* estimate was less accurate for *gzip* than for other benchmarks. *Gzip* is notably more compute-bound than the rest of the benchmarks, and the *R_IPC* equation did not suit it as well as the others. The model for *R_IPC* could be made more accurate if parameterized by the workload's instruction mix. For the majority of the

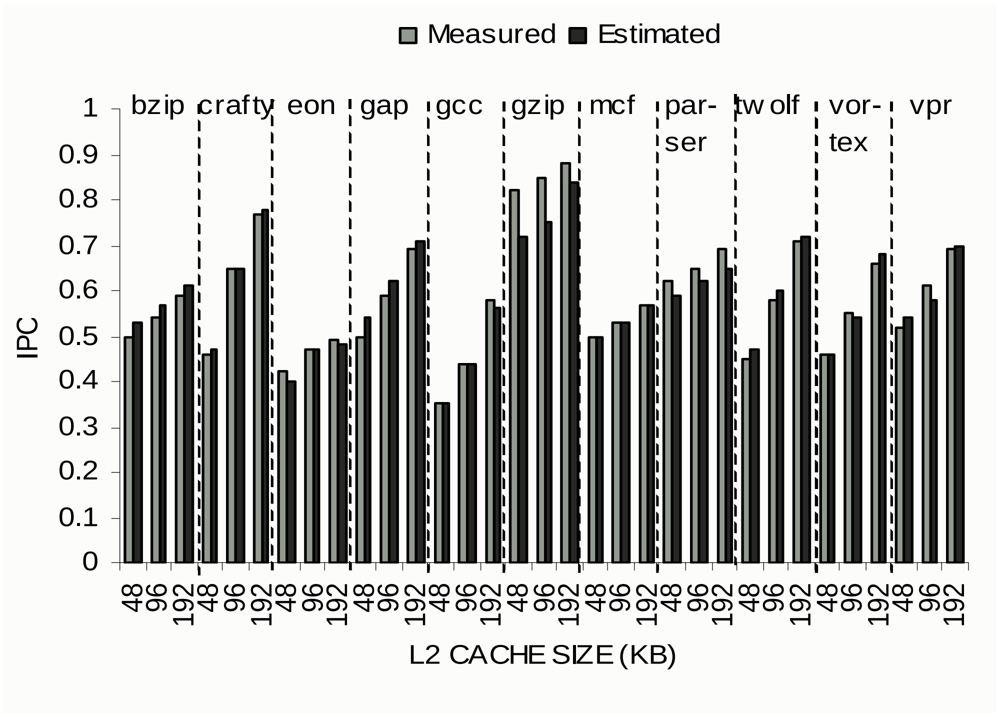


**Figure 3-2. Measured vs. estimated IPC with four concurrent threads**

**Table 3-1. Measured vs. estimated IPC for varying degrees of concurrency and a 48KB L2 cache**

| | **4 threads** | | | **3 threads** | | |
|---|---|---|---|---|---|---|
| | Measured | Estimated | Difference | Measured | Estimated | Difference |
| bzip | 0.50 | 0.53 | 6.00% | 0.64 | 0.67 | 4.69% |
| crafty | 0.46 | 0.47 | 2.17% | 0.40 | 0.49 | 22.50% |
| eon | 0.42 | 0.40 | 4.76% | 0.34 | 0.35 | 2.94% |
| gap | 0.50 | 0.54 | 8.00% | 0.46 | 0.49 | 6.52% |
| gcc | 0.35 | 0.35 | 0.00% | 0.31 | 0.38 | 22.58% |
| gzip | 0.82 | 0.72 | 12.20% | 0.68 | 0.71 | 4.41% |
| mcf | 0.50 | 0.50 | 0.00% | 0.18 | 0.20 | 11.11% |
| parser | 0.62 | 0.59 | 4.84% | 0.49 | 0.56 | 14.29% |
| twolf | 0.45 | 0.47 | 4.44% | 0.38 | 0.35 | 7.89% |
| vortex | 0.46 | 0.46 | 0.00% | 0.37 | 0.43 | 16.22% |
| vpr | 0.52 | 0.54 | 3.85% | 0.45 | 0.45 | 0.00% |
| | | Mean | **4.21%** | | Mean | **10.29%** |
| | | Median | **4.44%** | | Median | **7.89%** |
| | | Maximum | **12.20%** | | Maximum | **22.58%** |
| | | | | | | |
| | **2 threads** | | | | | |
| | Measured | Estimated | Difference | | | |
| bzip | 0.89 | 0.74 | 16.85% | | | |
| crafty | 0.34 | 0.41 | 20.59% | | | |
| eon | 0.26 | 0.27 | 3.85% | | | |
| gap | 0.43 | 0.46 | 6.98% | | | |
| gcc | 0.23 | 0.28 | 21.74% | | | |
| gzip | 0.52 | 0.55 | 5.77% | | | |
| mcf | 0.15 | 0.16 | 6.67% | | | |
| parser | 0.35 | 0.42 | 20.00% | | | |
| twolf | 0.32 | 0.29 | 9.38% | | | |
| vortex | 0.27 | 0.31 | 14.81% | | | |
| vpr | 0.35 | 0.35 | 0.00% | | | |
| | | Mean | **11.51%** | | | |
| | | Median | **9.38%** | | | |
| | | Maximum | **21.74%** | | | |

benchmarks, however, the simple *R_IPC* equation works well, as we expected.

Next we analyze the model's sensitivity to the number of concurrent threads. We set the cache size to 48KB and vary the number of concurrent threads $N$ from two to four. Our model requires *perf_cache_IPC(N)* as input. For $N = V$ we measure *perf_cache_IPC(V)* on our simulator. A scheduler would not be able to measure this quantity directly, but would estimate it (we explain how in Section III.3). For $N < V$ we do not measure *perf_cache_IPC(N)* directly, but estimate it, just as this would be done in a scheduler. (We use the equation derived in a similar fashion as the equation for *R_IPC*, as we will explain in Section III.3.)

Table 3-1 shows the data. The accuracy is comparable to that of the previous multithreaded IPC models: the mean, median and maximum errors reported by Saavedra-Barrera [62] are 7%, 4% and 28% respectively.

Our model is less accurate when the degree of concurrency is smaller than the number of virtual processors ($N < V$), because in this case, we estimate *perf_cache_IPC(N)* rather than measuring it directly.

A precise model of *perf_cache_IPC(N)* would account for architectural details of the processor and the characteristics of the workload. Ongoing work on modeling performance of multithreaded processors [40,55,68] suggests alternative avenues for approximating *perf_cache_IPC(N)* without sacrificing the model's simplicity.

### III.2.4 Accounting for the memory bus delay

Finally, we account for memory-bus delay. Existing models for memory-bus delay [70,77] are typically based on mean-value analysis of closed queuing networks [43,58]. We designed an alternative, simpler model. Appendix IX.1 presents the model in detail and shows how we incorporate the memory-bus delay in the IPC model.

Figure 3-3 shows how the IPC estimated using our memory-bus-delay enhanced model compares with the IPC measured on a simulated machine configured with memory bandwidth of 5.2GB/s. The differences between the measured and estimated values are 11% (mean), 10% (median), and 31% (largest).

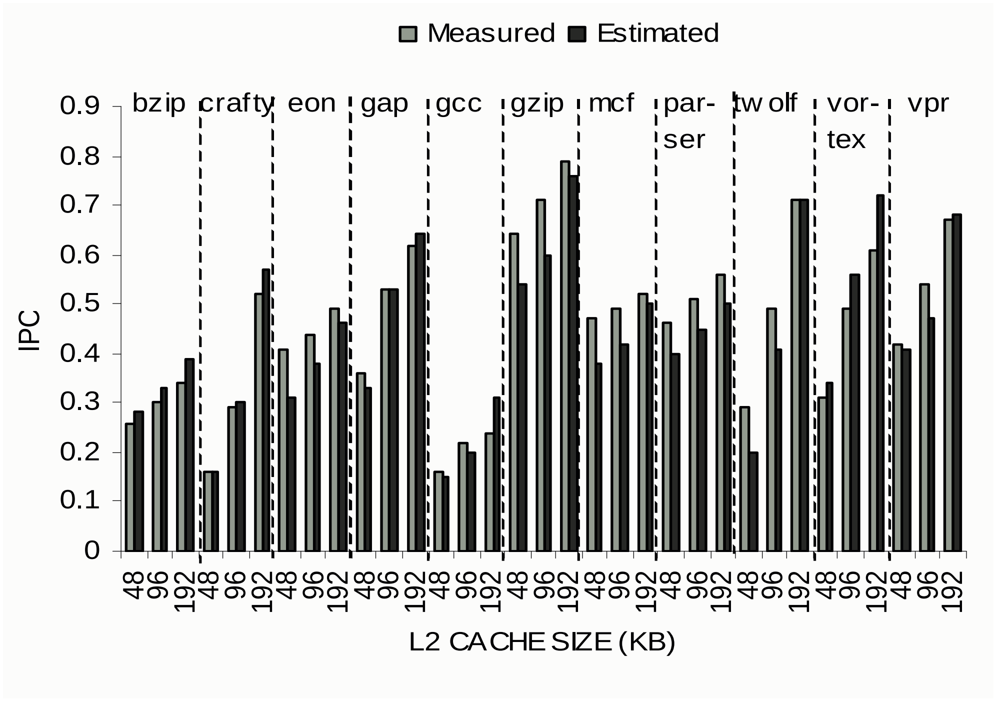


**Figure 3-3. Measured vs. estimated IPC with limited memory bandwidth**
The differences between measured and estimated values are 11% (mean), 10% (median), and 31% (largest).

## III.3 The Scheduler

Recall that we designed our model to use it in a non-work-conserving scheduler to determine the degree of concurrency producing the highest IPC. We built a non-work-conserving scheduler prototype to evaluate our algorithm. The scheduler runs at user level and starts a set of specified jobs, spawning a thread for each job.

The scheduler works in two phases: the *preparation* phase and the *optimization* phase. During the preparation phase it measures the runtime statistics that it later feeds to the model to estimate the optimal IPC. During the optimization phase, the scheduler estimates the optimal IPC and dynamically configures the processor to use the optimal degree of concurrency.

*Preparation phase:* The scheduler runs a multi-process or a multithreaded job using all $V$ virtual contexts, i.e. using a maximum degree of concurrency. It measures runtime statistics using hardware performance counters and generates inputs for the model, explained in more detail in Section III.3.1. This phase lasts until each thread retires roughly 100 million instructions – we empirically determined that for our benchmarks this length is sufficient to capture long-term cache-locality properties (Appendix IX.3).

*Optimization phase:* For each degree of concurrency $N < V$, the scheduler estimates the IPC using our model. It configures the processor to use the degree of concurrency corresponding to the highest predicted IPC by disabling one or more virtual processors. If the IPC obtained with the maximum degree of concurrency $V$ (measured during the preparation phase) is higher than any of the estimated IPC for $N < V$, the scheduler keeps all virtual processors busy.

The optimization phase lasts as long as the workload exhibits similar cache locality properties. If a workload enters a new locality phase, the optimization phase must be stopped, and the preparation phase must be repeated. We analyzed temporal caching behavior of our benchmarks and found that most of our benchmarks show little variation in caching behavior over time (Appendix IX.3). For such benchmarks, it is sufficient to repeat scheduling phases at a fixed interval, such as every one billion instructions (roughly once a second). For benchmarks with frequently changing access patterns, such as *gcc*, changes in locality phases must be detected dynamically, and the scheduling phases must be repeated when a change is detected. Previous research showed how to perform dynamic online locality phase detection on conventional processors [16,64]. Similar algorithms must be designed for multithreaded processors. Our current prototype does not repeat the preparation phase, assuming that the workload does not change its cache locality properties throughout the execution. As the data in Appendix IX.3 show, this assumption holds for most of our benchmarks.

## III.3.1 Obtaining the input parameters for the model

The scheduler estimates the IPC for each degree of concurrency *N*,

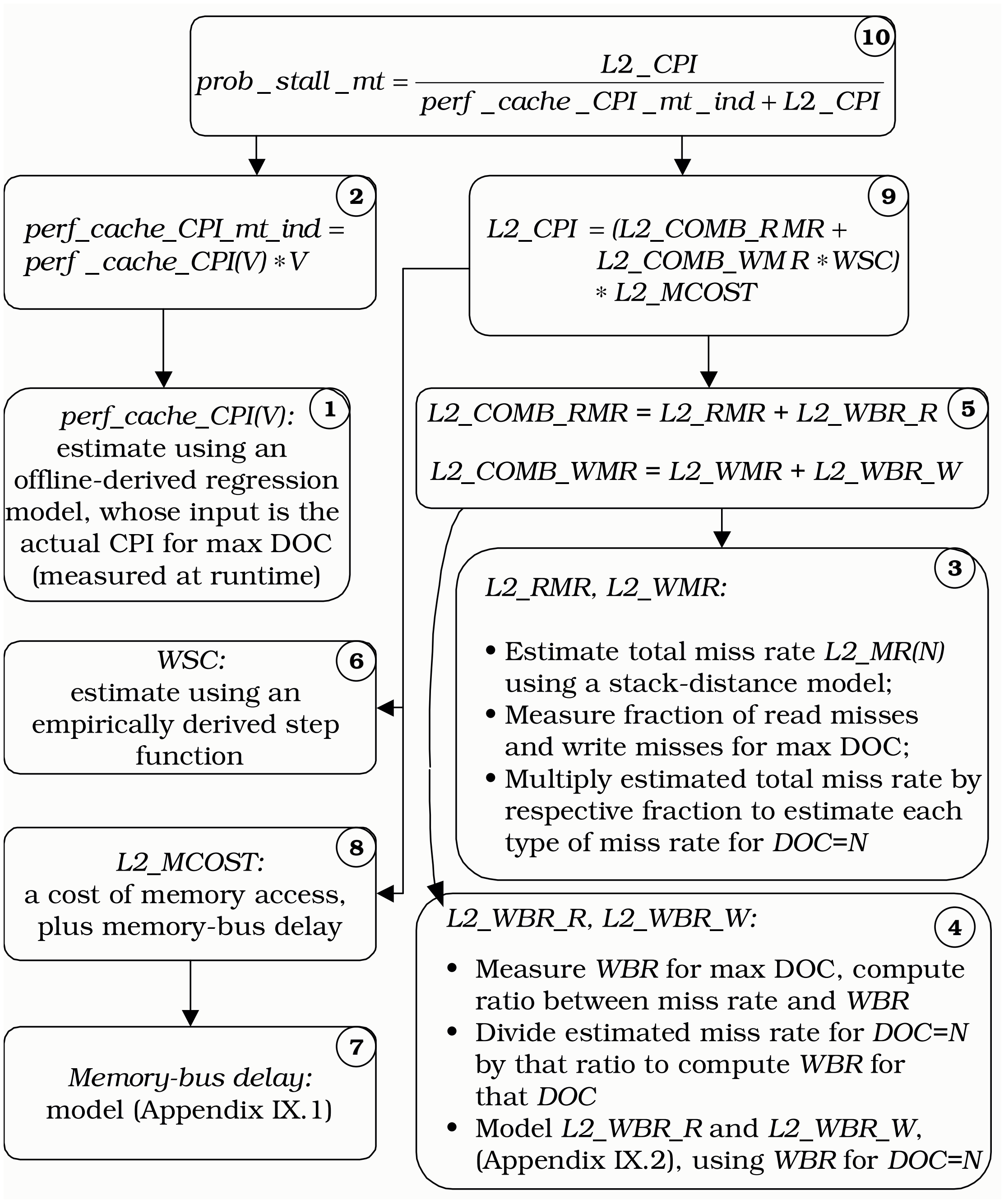

10
$prob\_stall\_mt = \frac{L2\_CPI}{perf\_cache\_CPI\_mt\_ind + L2\_CPI}$
2
$perf\_cache\_CPI\_mt\_ind = perf\_cache\_CPI(V) * V$
9
$L2\_CPI = (L2\_COMB\_RMR + L2\_COMB\_WMR * WSC) * L2\_MCOST$
1
perf_cache_CPI(V): estimate using an offline-derived regression model, whose input is the actual CPI for max DOC (measured at runtime)
5
L2_COMB_RMR = L2_RMR + L2_WBR_R
L2_COMB_WMR = L2_WMR + L2_WBR_W
3
L2_RMR, L2_WMR:
• Estimate total miss rate L2_MR(N) using a stack-distance model;
• Measure fraction of read misses and write misses for max DOC;
• Multiply estimated total miss rate by respective fraction to estimate each type of miss rate for DOC=N
6
WSC: estimate using an empirically derived step function
8
L2_MCOST: a cost of memory access, plus memory-bus delay
4
L2_WBR_R, L2_WBR_W:
• Measure WBR for max DOC, compute ratio between miss rate and WBR
• Divide estimated miss rate for DOC=N by that ratio to compute WBR for that DOC
• Model L2_WBR_R and L2_WBR_W, (Appendix IX.2), using WBR for DOC=N
7
Memory-bus delay: model (Appendix IX.1)

**Figure 3-4. Summary of the model**
Step numbers are shown in circles. Arrows indicate dependencies for each equation. Trace back the arrows to understand how we obtain the values for each equation. We use *DOC* to abbreviate *degree of concurrency.*

where $N < V$, using Equation III.8. The factors in this equation are *R_IPC(K)* and processor state probabilities *P(K).* We derive the equation for *R_IPC* (Equation III.9) offline, and supply it to the scheduler on start-up. Computing probabilities *P(K)* (Equation III.7) requires *prob_stall_mt.*

Figure 3-4 summarizes the steps used to estimate *prob_stall_mt.* We now elaborate on the steps used to obtain *perf_cache_IPC(N)* – the perfect-cache IPC for *N* threads (steps 1 and 2 in the figure), and *L2_MR(N)* – the L2 cache miss rate achieved by *N* concurrent threads (step 3 in the figure).

### *III.3.1.1 Obtaining the perfect-cache IPC*

The perfect-cache IPC achieved by multiple homogeneous threads depends on two factors: how efficiently these threads use the processor and how many threads are actually running. Therefore, the scheduler computes *perf_cache_IPC(N)* using a two-step process: (1) compute the perfect-cache IPC for the maximum degree of concurrency, *perf_cache_IPC(V).* This tells us how efficiently the threads use the processor; (2) For a given *N*, estimate *perf_cache_IPC(N)*, using a regression-derived linear equation. We describe these steps in more detail:

(1) We compute *perf_cache_IPC(V)* by applying the inverse of our model. As Figure 3-4 shows, our model (Equation III.8) is expressible as a function of the workload's perfect-cache CPI and L2 miss rate:

$$IPC = F(N, perf\_cache\_CPI(N), L2\_MR(N)).$$

We first compute *perf_cache_CPI(V)* by applying the inverse of the model:

$$perf\_cache\_CPI(V) = F'(V, IPC, L2\_MR(V)).$$

(The actual IPC and the L2 miss rate *L2_MR(V)* are measured during the preparation phase.) We then compute *perf_cache_IPC(V)* by taking the inverse of *perf_cache_CPI(V).*

(2) We estimate *perf_cache_IPC(N)* using a regression-derived linear equation, where *N* and *perf_cache_IPC(V)* are supplied as inputs. The equation has the same form and is derived in a similar way as the *R_IPC* equation (Equation III.9). We supply this equation to the scheduler upon initialization.

We evaluated the accuracy of the estimated *perf_cache_IPC(N)* and found that the difference between the actual and the estimated *perf_cache_IPC(N)* is 7.14% (mean), 6.17% (median), and 22.92% (max). Furthermore, we found that if we derive separate equations for high-to-medium-IPC and low-IPC workloads[5], the accuracy significantly improves: the difference between the actual and the estimated *perf_cache_IPC(N)* is 2.7% (mean), 1.55% (median), and 6.87% (max).

[5]We classified a workload as low-IPC if its perfect-cache IPC was below the 95% confidence interval for a mean perfect-cache IPC of the entire SPEC CPU2000 integer suite.

#### *III.3.1.2 Obtaining the L2 miss rate*

There are two known techniques to estimate *L2_MR(N)*, the L2 miss rate when $N < V$ threads run in parallel: the *stack-distance* model and the *reuse-distance* model.

A *stack distance* is the distance of a cached memory location from the top of the cache's LRU stack at the time that location is referenced. A frequently reused memory location will be characterized by short stack distances. We also speak of a *stack-distance profile* – a short summary of a program's stack distances. The stack-distance profile can be obtained statically using a compiler [21] or at runtime if the machine has appropriate hardware counters [73]. A single-threaded stack-distance model estimates cache miss rates for a thread using the corresponding program's stack-distance profile. A multithreaded stack-distance model combines multiple stack-distance profiles to estimate miss rates for $N$ concurrent threads [23].

A *reuse distance* is the number of distinct addresses referenced between two references to the same address. A *reuse-distance histogram* is a summary of reuse distances. A *reuse-distance* model [18] requires a thread's reuse-distance histogram to estimate that thread's miss rate. Reuse-distance histograms can be obtained online, but this process is expensive.

There are advantages and downsides to using each model. The stack-distance model is said to have poor accuracy for workloads with frequently changing execution patterns [21]. Additionally, it relies on compiler-generated stack distance profiles. A reuse-distance model is more flexible, but generates performance overhead, sometimes as much as 40% [18]. Although accuracy can be sacrificed for performance with the reuse-distance model, more work is needed to determine whether an acceptable trade-off can be achieved.

In our prototype, we estimated *L2_MR(N)* using a method based on the stack-distance model. Although we expect that a real scheduler would use compiler-generated stack-distance profiles, we did not have such a compiler, and so we instrumented our simulator to build stack-distance profiles. We generated profiles for all benchmarks in advance via simulation, and then supplied them to the scheduler.

The original stack-distance model [21] was designed to estimate cache miss rates on single-threaded processor architectures. Chandra et al. extended it to work for chip multiprocessor architectures with a shared L2 cache [23]. Their model compares stack-distance profiles for concurrent threads, and based on that comparison estimates the miss rate for each thread under cache sharing. We could not apply Chandra's model directly, because our simulated architecture has a shared L1 cache in addition to the shared L2 cache. Sharing of the L1 cache increases the L1 cache miss rate, generating additional references, and possibly misses, in the L2 cache. And this may cause a change in the thread's stack-distance profile for the L2 cache. To make the model suitable for the architectures with multiple levels of shared caches, we implemented it in our scheduler prototype as follows:

- At runtime, the scheduler derives the workload-specific *adjustment coefficient* – a value by which it multiplies the L2 cache miss rate estimated using Chandra's model to account for changes in the L2-cache stack distance due to sharing of the L1 cache. (We explain how we derive the coefficient below).
- For degrees of concurrency smaller then the maximum, the scheduler estimates the L2 miss rate using Chandra's model and multiplies it by the adjustment coefficient.

*Deriving the adjustment coefficient*: The scheduler measures the actual L2 cache miss rate achieved with the maximum degree of concurrency. It also obtains the estimated miss L2 miss rate for the maximum degree of concurrency using Chandra's model. It computes the adjustment coefficient by calculating the ratio of these two quantities.

### III.3.2 Performance results

In this section, we demonstrate how the non-work-conserving (NWC) scheduler improves application performance by determining the best degree of concurrency. We chose several memory-bound benchmarks (such benchmarks are likely to benefit from the non-work-conserving policy) and one compute-bound benchmark. Our experiment consists of running four concurrent instances of a benchmark on a single-core simulated processor with four virtual contexts. We use several cache sizes and run the SPEC benchmarks using reference input sets. We fast-forward the simulation until all the threads have reached their main processing loop and then perform detailed simulation for 100 million instructions.

The NWC scheduler performs the preparation phase during this execution segment. At the end of the preparation phase, the scheduler uses our model to determine the degree of concurrency yielding the highest IPC. To determine performance with the NWC scheduler, we measure the IPC achieved with the degree of concurrency selected as the best over a period of 100 million instructions.

To measure performance with the default scheduler, we simulate the workload using the maximum degree of concurrency and measure the IPC. We determine the optimal IPC by simulating each benchmark using all possible degrees of concurrency, and recording the highest IPC.

Figure 3-5 shows the results. There are three sets of bars for each experiment, showing the IPC achieved with the default scheduler (*default)*, the non-work-conserving scheduler (*NWC)*, and the optimal IPC

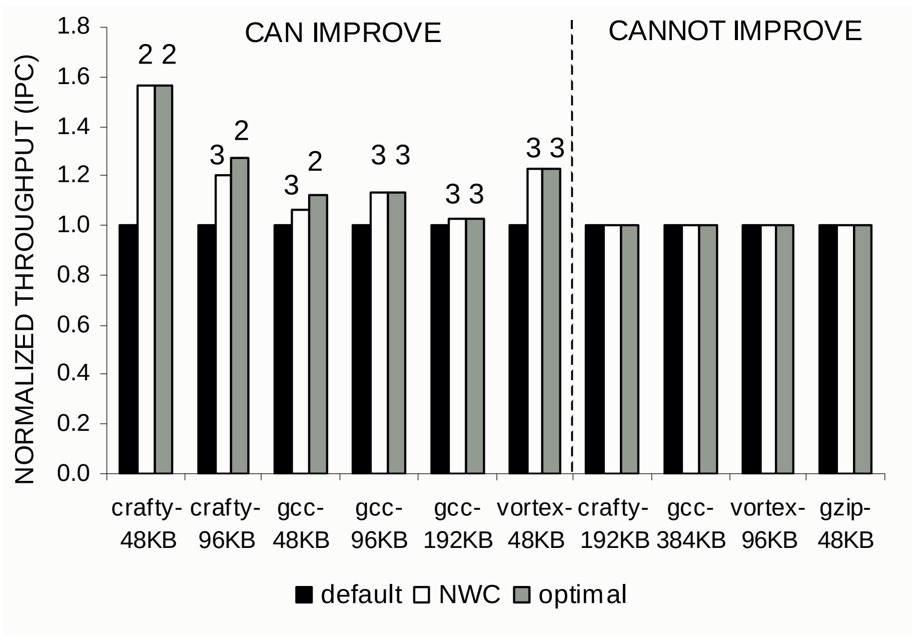


**Figure 3-5. Normalized throughput (IPC) with conventional operating system scheduler (default), with our non-work-conserving (NWC) scheduler, and the optimal IPC**
On the left side of the graph are the benchmark/L2 cache configurations where a non-work-conserving policy improves throughput (CAN IMPROVE). The numbers above the bars for NWC and optimal schedulers indicate the degree of concurrency used by that scheduler. On the right side of the graph are the configurations where the best throughput is already achieved with the maximum degree of concurrency (CANNOT IMPROVE).

*(optimal).* The left side of the graph shows the cases where the non-work-conserving policy achieves better performance (CAN IMPROVE). The NWC scheduler improves performance in each case, from 3% to 56%, often achieving the IPC as high as the optimal. The numbers above the bars for NWC and optimal schedulers indicate the degree of concurrency used by that scheduler.

On the right side of the graph, we show the cases where the non-work-conserving policy brings no performance improvement (CANNOT IMPROVE): the IPC with the default scheduler is already equal to the optimal. In these cases the NWC scheduler correctly decides to keep all virtual processors busy, achieving the optimal IPC. Note that each memory-intensive benchmark that appears in the "CAN IMPROVE" section of the graph also appears in the "CANNOT IMPROVE" section with a larger cache size, where performance can no longer be improved by lowering the degree of concurrency.

Our results demonstrate the feasibility of implementing a non-work-conserving scheduler that uses an online model to determine the best degree of concurrency. Although we have not evaluated the performance overhead generated by the scheduler, we believe it can be made small: most calculations use only simple integer arithmetic and hardware counter reads, which typically have small overheads. The key to obtaining these results is our simple model that works with inputs obtainable during compilation and at runtime.

## III.4 Summary

We presented a non-work-conserving scheduler for multithreaded processors and evaluated it using a user-level prototype. Our scheduler determines when using fewer than the maximum number of thread contexts improves performance. The scheduler employs a new online performance model, which is simple, but as accurate as previous off-line models. We achieved simplicity by precisely modeling contention for the L2 cache and approximating contention for other resources using simple models. To further enhance fidelity of our scheduler, it is worth improving online models for cache miss rates on multithreaded processors. It is also worth developing methods for online derivation of equations for perfect-cache IPC. Another avenue of future work is to apply our model to other multithreaded architectures and evaluate it using heterogeneous workloads.

# Chapter IV. THE TARGET-MISS-RATE ALGORITHM

## IV.1 Overview

The target-miss-rate algorithm reduces contention for the L2 cache by co-scheduling threads that collectively produce an L2 cache miss rate below a certain threshold. Unlike the non-work-conserving algorithm that adjusts the degree of concurrency, the target-miss-rate algorithm keeps all thread contexts busy. Keeping the cache miss rate under control ensures that processor utilization remains high.

In order to maintain a target cache miss rate the algorithm needs to identify the groups of co-scheduled threads satisfying the target miss rate. We call such thread groups *target-miss-rate sets*. To identify target-miss-rate sets, we model the cache miss rates for groups of threads using a new analytical model. The model works online and relies on hardware counters available on modern processors.

The target miss-rate algorithm is inspired by the *balance-set* scheduling algorithm, whose goal is to improve performance of virtual memory [27]. A *balance set* is a subset of processes whose combined working set fits in memory. Scheduling processes in balance sets assures that the system achieves high hit rates in the virtual memory system. While the operating system can estimate a thread's virtual memory working set, estimating the CPU-cache working set is challenging because cache allocation is performed automatically, by the hardware. Therefore, we do not use balance-set scheduling directly. Instead of scheduling balance sets, we schedule target-miss-rate sets: sets of threads satisfying the target cache miss rate.

We found that the target-miss-rate algorithm reduces the overall L2 cache miss rate and improves processor utilization, but does so at the expense of threads with high individual cache miss rates (*high-miss-rate threads*). Because high-miss-rate threads contribute more to the overall miss rate than low-miss-rate threads, they belong to a smaller number of target-miss-rate sets, and, as a result, get scheduled less often than low-miss-rate threads. This performance effect on high-miss-rate threads makes the algorithm poorly suited for situations when high-miss-rate threads are of high priority for the user. The target-miss-rate algorithm could be applicable to systems where predictable system performance is a high priority, such as real-time systems [13].

This chapter describes the design, implementation and evaluation of the target-miss-rate algorithm. Section IV.2 describes the cache model used in the algorithm, Section IV.3 describes the algorithm's design and its implementation in the Solaris operating system. Section IV.4 presents the evaluation of the algorithm, and Section IV.5 summarizes our findings.

## IV.2 Modeling Cache Miss Rates

Existing cache models are based on the *working-set* principle or the *reuse-distance* principle. Working-set models take as input the program's working set size, while reuse-distance models rely on a summary of the program's memory reuse patterns. The working-set model was not well suited for our purposes for two reasons: 1) the operating system cannot directly estimate a thread's cache working set, and 2) a program's working set size changes over time, and it is not trivial to determine the right methodology for estimating the working set size (previous working-set models provide no such methodology [10]).

We experimented with an existing reuse-distance model from Berg and Hagersten[6] [18] and observed high accuracy of estimated cache-miss-rates for workloads running on single-threaded processors. We adapted this model to work for multithreaded workloads on shared-cache systems. We next describe the reuse-distance model and how we adapted it for shared-cache processors.

### IV.2.1 The reuse-distance cache model

The reuse-distance model estimates cache miss rates using a thread's reuse distance profile. A reuse distance is the number of memory references made between two accesses to the same location. A

[6] Other similar models include a stack-distance model [21]. The stack-distance model relies on input from the compiler, so we did not consider it.

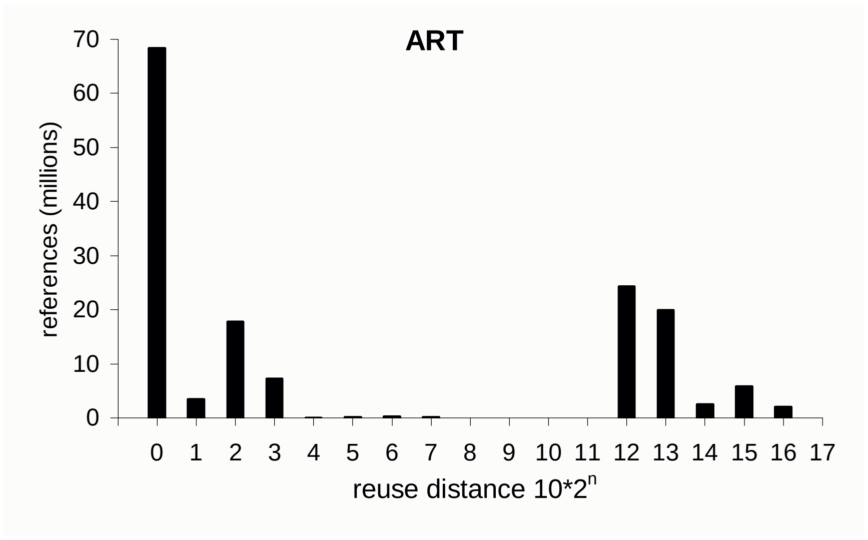
small reuse distance increases the chance of hitting in the cache: a recently referenced memory location is more likely to be in the cache than one referenced a long time ago. A reuse-distance profile captures the distribution of reuse distances. Figure 4-1 shows the reuse-distance profiles for two SPEC CPU2000 benchmarks *art* and *gzip.* We captured these reuse-distance profiles using our simulator instrumented to summarize memory references. We simulated each benchmark for 500 million instructions after reaching the main loop.

On the x-axis we display the reuse distance (reuse distance equals $10 \bullet 2^{x}$). On the y-axis we show the number of references (in millions) corresponding to each reuse distance. (The y-axis on the figure for *art* is larger than on the figure for *gzip*).

Note the difference between the shapes of these reuse-distance profiles. A significant portion of *art*'s references have a large reuse distance, while most of *gzip*'s references have a short reuse distance. The shapes of these profiles explains the high miss rate commonly reported for *art* and low miss rate reported for *gzip* [23].

Berg and Hagersten's reuse-distance model estimates cache miss rates using reuse-distance profiles. The idea is to compute, for each reuse distance, the probability that the memory references within that distance result in a miss, and then, to estimate the overall miss rate by computing the weighted average of these probabilities. The higher the reuse distance the higher the probability that a memory reference generates a cache miss.

Berg and Hagersten estimated cache miss rates of SPEC CPU2000



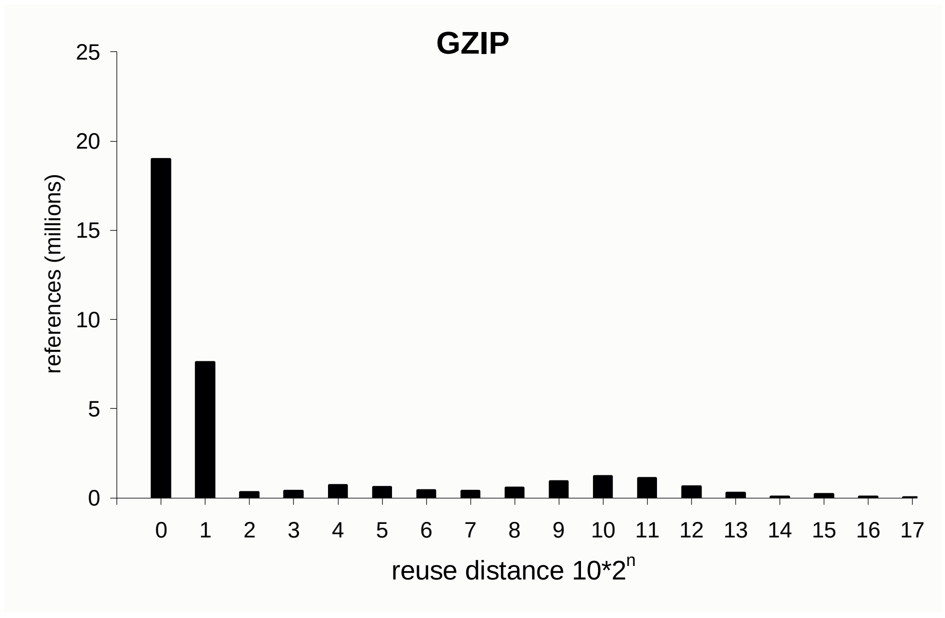


**Figure 4-1. Example reuse-distance profiles**
Reuse distance profiles for *art*, a benchmark typically exhibiting high cache miss rate, and *gzip*, a benchmark with low miss rate. (Note different scales on the y-axis).

benchmarks to within about 5% of the actual miss rates. We achieved similar accuracy with our workloads running on a single-threaded processor.

#### *IV.2.1.1 Adapting reuse-distance model for multithreading*

To implement the target-miss-rate algorithm, we need to model cache miss rates achieved by groups of co-scheduled threads. We express L2 cache miss rate as *percentage of L2 cache references resulting in misses*. To model the aggregate miss rate for groups of threads we adapted the Berg-Hargersten reuse-distance model to work for multithreaded workloads on shared-cache systems. We combine threads' individual reuse-distance histograms and estimate the aggregate miss rate using the combined reuse-distance profile.

We developed three methods for combining reuse-distance histograms: COMB, COMB+IPC, and AVG. COMB aggregates individual reuse distance histograms into a single histogram by summing up the references falling within the same reuse distance. COMB+IPC improves on COMB by making adjustments for the difference in the threads' instructions per cycle (IPC). AVG estimates the miss rate for each thread in a group as if the thread ran with a dedicated partition of a smaller cache and then averages the resulting miss rates. Although these methods are roughly equally accurate, the AVG incurs less runtime overhead, and is thus the method of choice in our algorithm. We now describe these methods in detail and evaluate their accuracy.

#### *IV.2.1.1.1 COMB*

COMB aggregates reuse-distance histograms using the following algorithm:

1. For each reuse distance, sum the number of references falling within that distance for all histograms.
2. Multiply the value of each reuse distance appearing in the histogram by a coefficient *C*.

Multiplying by a coefficient C in step 2 accounts for the increased reuse distance when several threads share the cache. Suppose a thread running on its own reused a memory location with a distance of one. When this thread runs with N-1 other threads, those other threads could make memory references between the first thread's references. As a result, from the point of view of the cache, the first thread's reuse distance of one, would become anywhere from one to N.

We experimented with setting the coefficient *C* to values from 1 to *N*, where *N* is the number of concurrent threads, and found that no single value worked well across the board. It was necessary to find a way to dynamically determine the right coefficient. Our next method

COMB+IPC dynamically computes the appropriate coefficient by taking into account the threads' relative instructions per cycle (IPC).

### *IV.2.1.1.2 COMB+IPC*

Recall that reuse-distance profiles capture how many memory references a thread makes between accesses to the same location. When threads share the cache, memory-reuse patterns of slower (i.e., low-IPC) threads have a smaller effect on the overall cache miss rate than memory-reuse patterns of fast threads, because slower threads will issue memory references at a slower rate. COMB+IPC takes this into account when aggregating reuse-distance histograms.

To understand how we adjust for relative differences in instructions per cycle (IPC), consider the following example: Suppose FastThread runs twice as fast as SlowThread – their respective IPCs are 1 and 0.5. When these two threads run together, about half of FastThread's references will be interspersed with SlowThread's

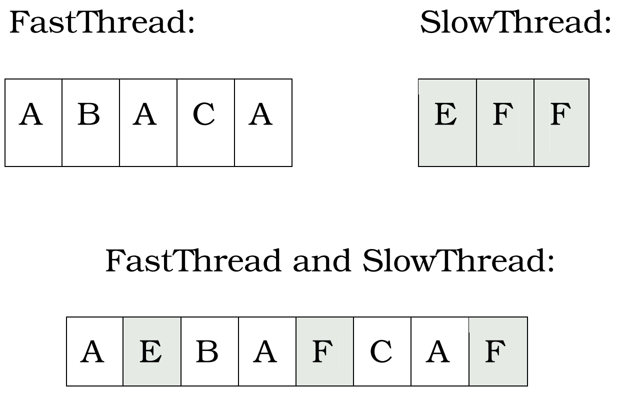


**Figure 4-2. Reuse distances change when threads with unequal speeds run in parallel**

references. Therefore, for FastThread, half of the reuse distances will remain the same, and the other half will increase by a factor of two (see Figure 4-2). To reflect this, we multiply the reuse distances in FastThread's histogram by the following coefficient: (0.5 * 1 + 0.5 * 2) = 1.5.

SlowThread will have all of its memory references interspersed with the references made by FastThread. In fact, each reference that SlowThread makes will be interspersed with two memory references made by FastThread. Therefore, all reuse distances will increase by a factor of three.

These observations lead us to the following formula for DIST_COEFF, a coefficient by which we adjust the values of reuse distances of each thread:

$$DIST\_COEFF_i = \left( \sum_{j=1}^{n} IPC_j \right) / IPC_i \ ,$$

where $i$ is the index of the thread whose coefficient we are computing and $n$ is the number of threads in the group.

Finally, we need to normalize the number of references to adjust for different lengths of time intervals we use for collecting the profiles. We collect reuse-distance profiles over the window of 500 million instructions. The time it takes to execute 500 million instructions depends on how quickly the thread runs. So a "slow" thread, running at the IPC of 0.5, will take twice as long to execute its 500 million instructions as the “fast” thread, running at the IPC of 1.0. During the time it takes the “fast” thread to execute its 500 million instructions, the “slow” thread will make half as many memory references than there are in its profile. We need to scale down the number of references in the “slow” thread’s profile to reflect that. We compute the normalization coefficient for the number of references *REFS_COEFF* as follows:

$$REFS\_COEFF_i = \frac{IPC_i}{MAX\_IPC},$$

where *i* is the index of the thread whose coefficient we compute and *MAX_IPC* is the IPC of the fastest thread.

We adjust each reuse distance histogram by multiplying reuse distances by *DIST_COEFF* and the number of references by *REFS_COEFF*. Then we simply merge all reuse distance histograms into one.

#### *IV.2.1.1.3 AVG*

We introduce the definition of a *partial cache miss rate* (PCMR) – the percent of L2 cache references that missed in the cache when a thread ran with a dedicated partition of the shared cache. The size of the partition equals ORIGINAL_CACHE_SIZE / NUM_THREADS. This is a key definition for the target-miss-rate algorithm.

The AVG method assumes that each thread runs with its own dedicated partition of the shared cache and estimates threads' PCMRs using reuse-distance profiles. Then it averages the PCMRs to estimate the aggregate miss rate for the group of co-scheduled threads.

#### *IV.2.1.2 Evaluation of the multithreaded model*

To evaluate the effectiveness of the model, we used an 18-thread SPEC CPU2000 workload, consisting of two copies of each of these nine benchmarks: *ammp, art, crafty, gcc, gzip, parser, vortex, vpr (place), vpr (route)*. We estimated miss rates for all possible groups consisting of eight threads, for four different L2 cache sizes, using COMB, COMB+IPC and AVG. We validated the estimated miss rates for several randomly selected groups – six for each cache size.

Figure 4-3 shows how the estimated cache miss rates compare against the actual miss rates. For COMB method we used the coefficient *C* producing the most accurate estimates. We measured the actual miss rates by simulating each validated thread group on our simulator configured with two four-way-multithreaded cores and varying L2 cache sizes. The miss rates shown in the figure are the mean of all groups for a given cache size (the coefficient of variance for a given cache size was always smaller than 30%).

The estimated miss rates are, on average, within 17% of the actual miss rates. The 95% confidence interval for the mean error is [10%,24%] for all models.

Chandra et al. adapted the stack-distance model (a variation of reuse-distance model) for multithreaded workloads on shared-cache systems, and the results of their work (carried out concurrently with ours) offer an opportunity for comparison. Chandra et al. presented three different models – two simpler ones, similar to our COMB and COMB+IPC, and one other more complex. Their simpler models produced roughly the same accuracy of estimated miss rates [23]as our model (they reported mean errors of 18.6% and 13.2% for these models). The more complex model yielded higher accuracy, with an average error of only 3.9%, but required a larger number of steps to compute and used

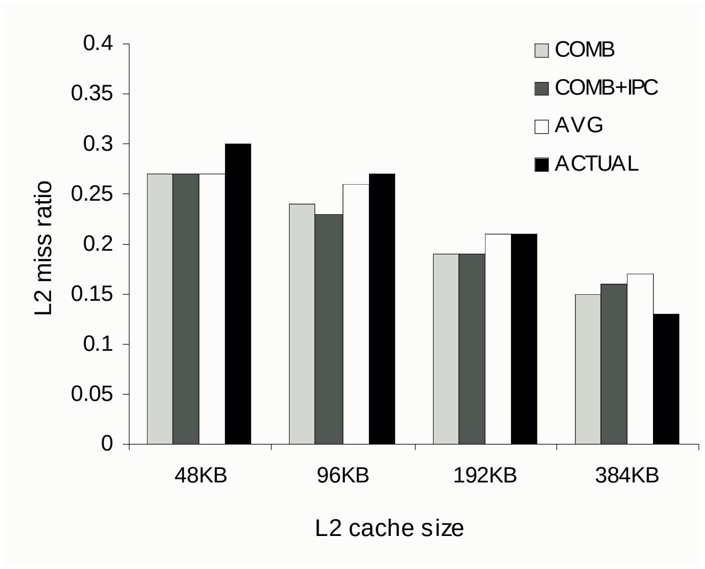


**Figure 4-3. Actual vs. estimated L2 miss rates with COMB, COMB+IPC and AVG**

floating-point operations, not permitted in many OS kernels. Given a higher complexity of this model it is not clear whether it would be suitable for use in an operating system scheduler.

There are several potential sources for error in our model. First, the single-threaded Berg-Hagersten model assumes a fully-associative cache, and we simulate a 12-way set associative cache. The stack-distance [21] model accounts for cache-associativity; so using a stack-distance model could improve accuracy. Another source of error is the assumption that cache misses are caused primarily by insufficient cache capacity. The presence of other kinds of misses, such as cold-start or conflict misses, contribute to the imperfect accuracy. While the effect of cold-start misses is minimal [10], modeling conflicts may improve accuracy. We are not aware of analytical cache models that account for conflict misses. Settle et al. designed a method to account for conflict misses when scheduling threads on multithreaded processors [63], however that method requires special hardware.

Of our three combining techniques, there was no clear winner in terms of accuracy. In the scheduling algorithm, we use AVG, because it can be implemented less expensively as it does not require aggregation of reuse-distance profiles. With AVG, estimating a miss rate for a group of threads requires a multiplication of those threads' PCMRs. The other two methods require aggregating the threads' reuse-distance profiles, and that is more costly than performing a set of multiplications.

## IV.3 Design and Implementation

We begin this section by discussing the challenges we encountered designing the target-miss-rate algorithm and how we addressed them. We then describe the algorithm's design and implementation.

### IV.3.1 Design challenges

The target-miss-rate algorithm schedules threads such that the overall miss rate generated by the workload remains below a target threshold level. It uses the AVG model to determine the miss rate generated by any group of threads and then ensures that those threads whose estimated aggregate miss rate is above the threshold are never co-scheduled. While the idea is straightforward, there are several issues that make a low-overhead scalable implementation of this idea challenging. These issues have to do with online generation of reuse-distance profiles and making scheduling decisions without global synchronization across processors. In the remainder of this section we explain how we addressed these issues.

#### *IV.3.1.1 Generation of reuse-distance profiles*

We need reuse-distance profiles to estimate cache miss rates for groups of threads. Berg and Hagersten described an online method for generating reuse-distance profiles that monitors reuse patterns for a subset of the memory locations accessed by a program [18]. Software-level monitoring is performed by read-protecting memory pages, and then recording accesses to memory locations on those pages when the system traps on a protection violation. Because this method requires handling traps, it generates runtime overhead often over 40%, as reported by the authors. Our preliminary analysis and back-of-the-envelope calculations confirmed that the overhead of trap-based memory monitoring would be too significant for an operating system scheduler.

An alternative implementation of the reuse-distance model, the stack-distance model [21], requires compiler-generated memory-reuse profiles, and so we did not consider using the stack-distance model.

###### *IV.3.1.1.1 Group elimination: an alternative to reuse-distance profiles*

To avoid the overhead associated with generating reuse-distance profiles, we designed a new online method that eliminates the need to generate those profiles. Recall that AVG estimates threads' PCMR, i.e. the miss rate with a dedicated cache partition, using reuse-distance profiles and then averages them to estimate the aggregate miss rate for co-scheduled threads. We designed an alternative way of estimating threads' PCMR without reuse-distance profiles. We refer to it as the *group elimination* method.

The group elimination method observes miss rates for groups of threads and then estimates each thread's PCMR based on how the aggregate miss rate changes depending on group membership.

We assume that the overall L2 miss rate *L2MR* incurred by a group of *n* co-scheduled threads with a cache of size *C* can be expressed as a linear combination of the threads' PCMRs with cache size *C/n*. Specifically, we assume that the L2 miss rate is the average of individual threads' PCMRs:

$$L2MR(C) = \frac{1}{n}\sum_{i=1}^{n} PCMR_i(C/n)$$

This is the assumption underlying the AVG method. Although it is not always true, because it implies that there are no conflict misses, as we demonstrated in Section IV.2 it produces sufficiently accurate miss rate estimates for our scheduler.

The aggregate miss rate *L2MR* for a group of *n* co-scheduled threads can be expressed as:

$$(x_o + x_1 + \ldots + x_{n-1}) / n = L2MR,$$

where $x_i$ is the PCMR of thread *i*. Recall that we can measure *L2MR* at runtime on modern processors [5,8,37]. Observing *L2MR* for different groups of threads allows us to create a system of equations, which we then use to compute all $x_i$. The following matrix shows an example system of linear equations for a system with four co-scheduled threads.

$$\begin{array}{lcccccccl}
( & & x_1 & + & x_2 & + & x_3 & + & x_4)/n = L2MR(0) \\
(x_0 & & & + & x_2 & + & x_3 & + & x_4)/n = L2MR(1) \\
(x_0 & + & x_1 & & & + & x_3 & + & x_4)/n = L2MR(2) \\
(x_0 & + & x_1 & + & x_2 & & & + & x_4)/n = L2MR(3) \\
(x_0 & + & x_1 & + & x_2 & + & x_3 & & )/n = L2MR(4)
\end{array}$$

The group elimination method removes the need to generate reuse-distance profiles. We can implement the group-elimination model inside the operating system with low runtime overhead, using hardware counters available on modern processors. The group elimination method used with the AVG model allows us to estimate L2 miss rates for groups of co-scheduled threads to within 30% of the actual value. The accuracy is lower than if we used AVG with reuse-distance profiles, but we avoid the overhead of generating reuse-distance profiles.

#### IV.3.1.2 Co-scheduling without global synchronization

Recall that the target-miss-rate algorithm must co-schedule only those threads whose aggregate L2 cache miss rate is under the target threshold. Therefore, a scheduling decision requires an "agreement" between the system's virtual processors about the threads that they schedule. Schedulers on modern operating systems are written with per-processor scheduling queues, such that each processor makes scheduling decisions independently, simply pulling a thread off of its queue and stealing a thread from another processor's queue if necessary [54]. This design reduces inter-processor communication allowing for greater scalability [20]. Any co-scheduling design requiring communication among the virtual processors has inherent scalability limitations.

To maintain a scalable design inside our scheduler, we use an approach that does not require inter-processor communication while making scheduling decisions. We assign to each processor a maximum PCMR value MAX_PCMR. (Remember that a high PCMR means worse caching behavior and a low PCMR means better caching behavior.) Then we schedule a thread only on those processors whose MAX_PCMR exceeds or equals that thread's PCMR. Recall that we estimate the system's aggregate L2 miss rate by averaging the PCMRs of co-scheduled threads. By assigning MAX_PCMRs to processors such that their average is below the target threshold miss rate, we assure that the system's miss rate is always below the target threshold.

Figure 4-3 illustrates our approach. There are four virtual processors, *Processor 0* through *Processor 3*. Their respective MAX_PCMR values are 1, 5, 11 and 17. The PCMR of the thread running on a processor will never exceed the MAX_PCMR of that processor. The average of MAX_PCMRs is 8.5, so we expect that the system's L2 miss rate will not exceed 8.5%.

We also show, above each processor, the threads that can run on that processor. Thread *t0* has a PCMR of 1, meaning that *t0*'s estimated miss rate with a dedicated partition of a shared cache is 1%. *T0* can run on any processor, because its PCMR does not exceed any of the MAX_PCMRs. Thread *t3*, on the other hand, has a PCMR of 16, exhibiting poor caching behavior, and it can run only on *Processor3*, whose MAX_PCMR is 17.

In the implementation section of this chapter we describe the algorithm we use to assign MAX_PCMR to processors. We also describe the *thread CPU preferences mechanism* – the mechanism for forcing threads on processors with the right MAX_PCMR.

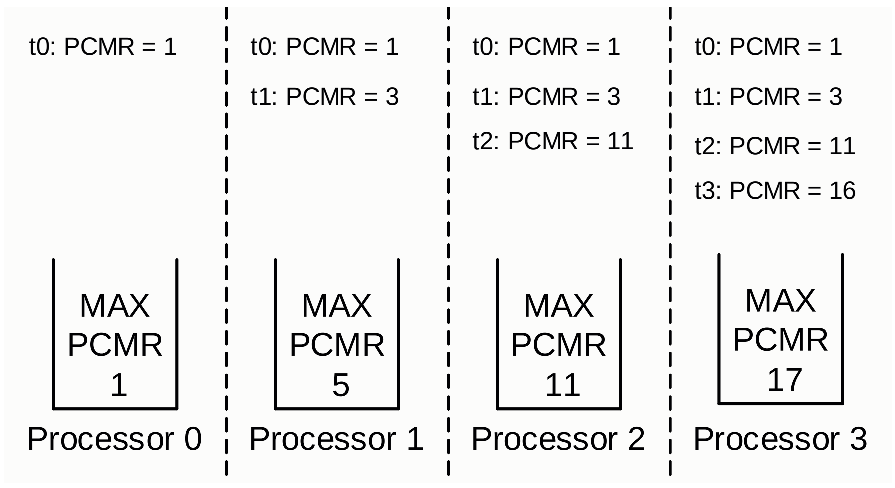


**Figure 4-3. An assignment of maximum PCMR to processors**

### IV.3.2 The structure of the algorithm

The algorithm operates in two phases: the measurement phase and the scheduling phase. During the measurement phase the scheduler estimates threads' PCMRs. It measures the L2 cache miss rates for groups of threads running concurrently until it has enough observations to create a system of linear equations, and then it solves the system to compute the threads' PCMRs.

During the scheduling phase, the scheduler assigns MAX_PCMRs to processors such that the average of all MAX_PCMR is below the target miss rate threshold. Then it launches the thread CPU preference mechanism so that threads are scheduled on processors with the "right" MAX_PCMRs.

### IV.3.3 Implementation

We begin with an overview of how the target-miss-rate algorithm fits in the Solaris 10 kernel scheduler. We then describe the implementation of the algorithm.

#### *IV.3.3.1 Target-miss-rate scheduling algorithm in Solaris*

We implemented the target-miss-rate scheduling algorithm in Solaris 10 as a loadable module. Modularity is a built-in feature of the Solaris scheduler. The scheduler consists of a *dispatcher* and a collection of *scheduling classes* [54]. A dispatcher implements the scheduling machinery: it places threads on processor queues, dispatches them on processors and enforces priorities within the queues. A scheduling class implements a scheduling policy: it assigns CPU time quanta and priorities and pre-empts threads when their time quantum expires. Solaris 10 provides several scheduling classes, each implemented in its own module, allowing a user to choose the one appropriate for his or her goals. Applications can use the target-miss-rate scheduling algorithm by invoking a `priocntl()` system call with an argument specifying the name of the module.

A Solaris scheduling class implements the scheduler interface. The kernel interacts with the scheduling module via this interface. Here are some of the functions defined in this interface:

| | |
|---|---|
| `cl_enterclass` | called when a thread enters the scheduling class; |
| `cl_tick` | called on every clock tick (1ms to 10ms); |
| `cl_sleep` | called when a thread blocks; |
| `cl_preempt` | called when a thread is preempted; |
| `cl_wakeup` | called when a thread unblocks; |
| `cl_exit` | called when a thread exits. |

For implementation of the group-elimination model, we needed the kernel to alert the scheduler whenever a thread is switched on and off a processor. We extended the scheduler interface with two new functions: `cl_onproc` and `cl_offproc`, and added the corresponding callbacks to the core kernel.

### *IV.3.3.2 Implementation of the measurement phase*

During the measurement phase, the kernel estimates threads' PCMRs. When there are no threads in the system with known PCMRs, the scheduler must observe enough thread groups to construct the system of linear equations for the group-elimination method.

Once there are enough threads in the system, we force the groups of threads, whose unknown PCMRs we want to represent in the system of equations, to run concurrently, so we can measure their aggregate miss rates. We want each group to stay on processor long enough so we can capture the representative cache miss rate of this group. To facilitate this, we elevate priorities of the threads in the group whenever a new group of threads begins running concurrently. Once the group has run for a period of time equal to MIN_RUNLENGTH (which we set to 1.8 million cycles after experimental evaluation of phase behavior of our benchmarks), we record the observed L2 miss rate for the group and restore priorities of the threads to their normal values. When a new group of threads appears on processor, we repeat the process. We continue until we have recorded the L2 miss rates for all thread groups represented in the system of linear equations. After that, we solve the system of equations, and estimate all threads' PCMRs.

When we have already estimated most threads' PCMRs, we can simplify the estimation of PCMRs for new threads entering the system.

We schedule a new thread, whose PCMR we do not yet know, with threads whose PCMRs we already know, and then we find the unknown PCMR by solving one linear equation. For example if we want to estimate the PCMR of a new thread *tx*, and we know the PCMRs of threads *t0*, *t1* and *t2*, we co-schedule those four threads, measure their aggregate L2 miss rate, and then compute the PCMR of *tx*.

###### *IV.3.3.2.1 Re-estimating the PCMRs*

Programs go through phases of execution [16,28,64,65]. A thread's PCMR may change when the thread enters a new execution phase. In that case, the scheduler needs to re-estimate the PCMRs. Our current implementation does not re-estimate the PCMRs, assuming that programs do not significantly change the L2 miss rate as they go through execution phases. In Appendix IX.3 we discuss this assumption and support it with data.

Enhancing our scheduling algorithm to re-estimate PCMRs would be a minor addition. A scheduler could choose a static frequency interval after which it would re-estimate the PCMRs, or it could dynamically detect phase changes and then re-estimate the PCMR. Because in our algorithm it is inexpensive to re-estimate the PCMRs, it would be reasonable to set a short value for a static interval, and re-estimate the PCMRs frequently. Another possibility is to re-estimate a program's PCMRs after dynamically detecting a program's phase change. Although there are online phase-detection algorithms for conventional, single-threaded, processors [16,64], we are not aware of such an algorithm for chip multithreaded processors. There are phase detection algorithms based on architecture-independent metrics, but they are expensive to use online [28,65].

#### *IV.3.3.3 Implementation of the scheduling phase*

When the measurement phase is complete, the system knows the PCMRs for most or all of the threads that it manages. The system schedules threads such that their aggregate miss rate remains below the target threshold value. Recall that we maintain a scalable design by assigning MAX_PCMR values to processors and then scheduling threads on those processors whose MAX_PCMR exceeds or equals the PCMR of the thread. In this section we discuss the following implementation details: how we select the target miss rate threshold, how we assign MAX_PCMRs to processors, and how we schedule threads on processors with the "right" MAX_PCMR.

##### *IV.3.3.3.1 Selecting the target miss rate threshold*

A target miss rate threshold could be a part of system configuration, and one way to determine a good value is via experimenting with different threshold values using workloads of interest. Our algorithm ensures that it does not completely starve threads, even if doing so violates the target miss rate threshold.

Therefore, it is safe to err on the low side when choosing the target miss rate threshold.

### *IV.3.3.3.2 Assigning MAX_PCMR to processors*

A MAX_PCMR assignment ensures that the average of all MAX_PCMRs does not exceed the target miss rate threshold unless satisfying this threshold would completely starve a thread. It also ensures that processors are never left idle.

We begin by assigning to each processor the maximum possible MAX_PCMR, i.e., the largest PCMR assigned to a thread in the system. Then we compute the projected L2 miss rate by averaging the MAX_PCMRs across all processors. If the projected miss rate is higher than the threshold, we decrease the MAX_PCMR for one of the CPUs. We repeatedly decrease MAX_PCMRs until the projected miss rate satisfies the target threshold.

To avoid completely starving any threads, we always ensure that a thread can run on at least $N$ processors on the system, where $N \geq 1$. This means that there are at least $N$ processors whose MAX_PCMR equals or exceeds the largest PCMR assigned to a thread in the system, even if this means that the projected miss rate is above the target threshold.

We also ensure that no processor's MAX_PCMR is below the smallest PCMR in the system, otherwise we would not be able to schedule any thread on that processor, leaving the processor unused.

#### *IV.3.3.3.3 Enforcing thread CPU preferences*

To ensure that a thread runs only on those processors whose MAX_PCMR exceeds or equals the PCMR of that thread, we devised a new mechanism in the Solaris kernel that we call *thread CPU preferences*. We associate with a thread a *CPU preference record*: the list of processor IDs corresponding to the processors where that thread can run. For our algorithm, a thread's CPU preference record contains those processors whose MAX_PCMRs equal or exceed the PCMR of that thread. Going back to Figure 4-3, thread *t0* would have processors 0, 1, 2 and 3 in its CPU preference record, and thread *t2* would have processors 2 and 3. We modified the Solaris dispatcher to dispatch a thread only to processors present in that thread's CPU preference record.

Solaris provides two other mechanisms that force threads to particular processors [54], but neither of them fit our needs. One mechanism is *processor binding,* which permanently binds a thread to a particular processor. This mechanism was not appropriate for us, because we needed the ability to schedule threads on a subset consisting of multiple processors. Another mechanism is specifying load groups consisting of multiple processors, and assigning a thread to a load group. This mechanism was not sufficiently flexible for us, because load groups cannot not be hierarchical or inclusive of other groups. So we could not, for example, create two load groups, such that one includes processors 0 through 3, and another includes processors 2 and 3.

Our thread CPU preference mechanism is low-overhead and adds only a dozen lines of code to the Solaris dispatcher.

## IV.4 Evaluation

We evaluate the target-miss-rate algorithm on a simulated UltraSPARC T1 processor and show that it reduces the overall system miss rate and improves throughput, but has the downside of reducing the processor share of high-miss-rate threads. We first describe the experimental configuration and the workload and then present the experiments. We conclude this section by analyzing the applicability of our results to a wide range of systems and workloads.

### IV.4.1 Experimental configuration

We configure our simulated processor with two UltraSPARC T1 cores running at 992 MHz, each equipped with four thread contexts. We perform experiments using L2 cache of sizes between 48KB and 768KB.

Since we use a full-system simulator we were able to install our own version of Solaris on the simulated hardware. Our version of Solaris includes the target-miss-rate algorithm. The operating system is not aware that it is running on the simulated hardware, so the changes to the operating system are identical to those that would be required for experiments on real hardware.

We set the target miss rate threshold to 15% for all cache sizes.

### IV.4.2 The workload

Our experimental workload consists of three applications: two SPEC CPU2000 benchmarks *twolf* and *art,* and the BerkeleyDB application reading records from a 100MB database in a hot-set pattern (80% of accesses are to 30% of the database). We chose these benchmarks because they represent a variety of caching behaviors and do not change L2 cache miss rate over time (and we have not yet adapted our algorithm for dynamic phase changes). We run six copies of each benchmark, creating a multiprogram workload of eighteen threads.

### IV.4.3 The experiment

We start our eighteen-thread workload on our experimental system and let the simulation continue through the measurement phase, when the scheduler estimates the threads' PCMRs. Once the measurement phase is complete, we begin measuring the system miss rate and the throughput achieved during the scheduling phase. The duration of the simulation in the scheduling phase is 16 billion machine instructions. We compare the miss rate achieved by the same workload with the default Solaris time-sharing scheduler and the target-miss-rate scheduler.

## IV.4.4 The results

In this section we present experiments demonstrating improvements in the overall cache miss rate and throughput, performance effects on high-PCMR threads, and runtime overhead.

### *IV.4.4.1 Improvements in the L2 cache miss rate*

Figure 4-4 shows the L2 cache miss rate achieved with the default scheduler (black bars) and with the target-miss-rate scheduler (white bars). The target-miss-rate algorithm reduces the overall miss rate by 15-46%, producing a throughput (instructions per cycle) improvement of 28-50% (Figure 4-5).

These results demonstrate that the scheduler was able to identify the threads that contributed most to the overall miss rate and to decrease the aggregate miss rate by limiting those threads' CPU

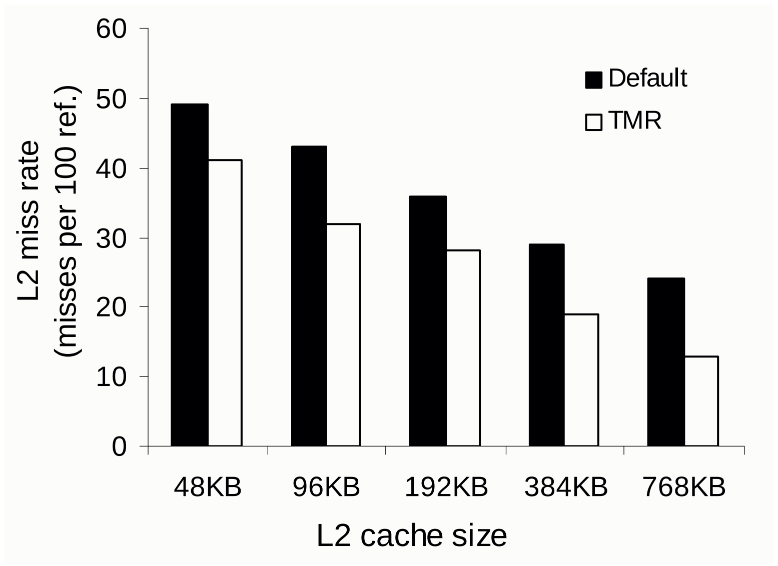


**Figure 4-4. Miss rate improvement with the target-miss-rate algorithm**

preferences. The algorithm achieved the largest miss rate improvement (of 46%) in the experiment with the 768KB L2 cache, because the algorithm identified all six threads running *art* (the benchmark with the highest individual miss rate) as high-PCMR threads. In other experiments, the scheduler sometimes failed to identify the threads running *art* as high-PCMR, so the improvements were smaller.

We found that in order to improve accuracy of PCMR estimations during the measurement phase it helps to discard cache-miss-rate measurements for short time intervals, because those measurements may be polluted with transition effects, i.e., a thread is renewing its working set in the cache after being off the processor. We discard L2 cache miss rate measurements for intervals shorter than 200μs.

We observed that the scheduler most frequently made a wrong

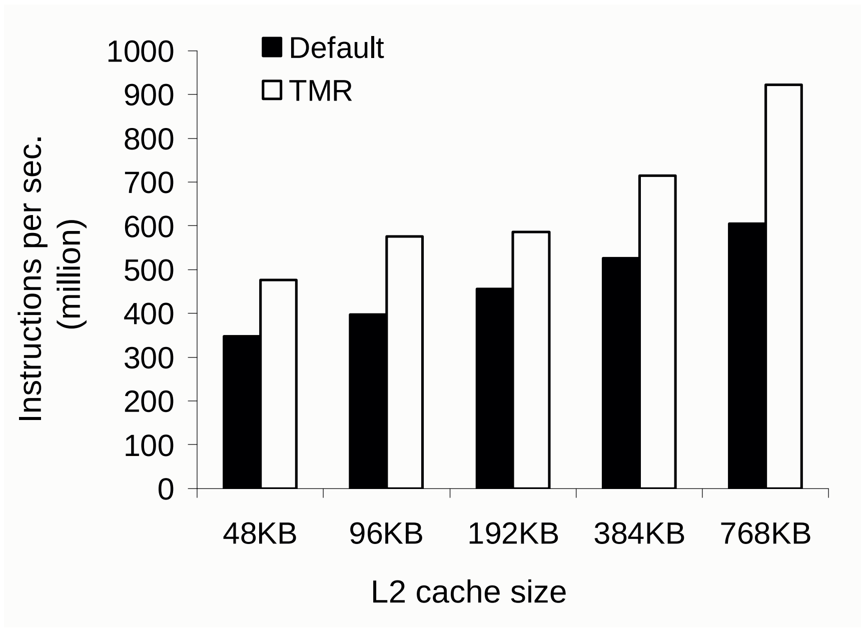


**Figure 4-5. Throughput improvement with the target-miss-rate algorithm**

PCMR assignment (i.e., giving a low PCMR to a high-miss-rate thread) during the part of the measurement phase where the scheduler had already assigned most PCMRs and ran the thread with only one group of co-runners to determine its PCMR. Using multiple co-runner groups could improve accuracy of the estimated PCMR in this case.

#### *IV.4.4.2 Miss Rate Accuracy*

We now compare the expected and the actual miss rate the system achieves with the target-miss-rate algorithm. We compute the expected miss rate using the MAX_PCMR assignments produced by our algorithm: recall that the expected overall miss rate is the average of MAX_PCMRs assigned to processors.

Figure 4-6 shows the results. Most expected miss rate values are within 25% of the actual miss rates (for cache sizes 96-768KB). For the

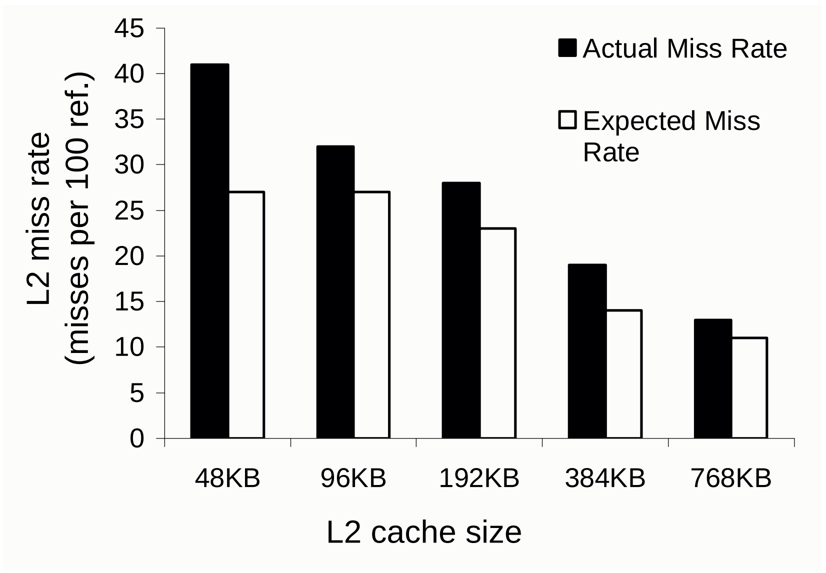


**Figure 4-6. Actual vs. expected miss rate with the target-miss-rate algorithm**

48KB cache, the expected miss rate is 34% below the actual miss rate. We note that the scheduler consistently underestimates the overall system miss rate. Investigating the source or error is a worthwhile direction of future work.

### *IV.4.4.3 Performance impact on high-PCMR threads*

The target-miss-rate algorithm improves the overall system miss rate at the expense of threads with high PCMR values. High-PCMR threads can run only on a handful of processors, unlike low-PCMR threads.

Figure 4-7 shows thread processor-time distribution with the default scheduler and the target-miss-rate scheduler with the 768KB L2 cache. Each box corresponds to the fraction of processor time allotted to a thread. Light-gray boxes correspond to high-PCMR *art* threads, dark-gray boxes to medium-PCMR *twolf* threads, and white boxes to low-

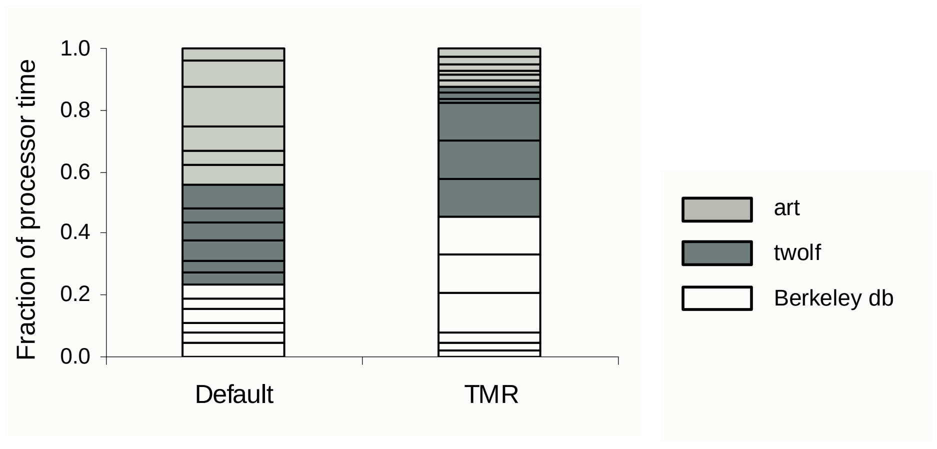


**Figure 4-7. Effect on processor time distribution**

PCMR Berkeley DB threads.

With the default scheduler, processor time distribution is more uniform than with the TMR scheduler. (Note that because the default scheduler uses the time-sharing policy, we cannot expect perfectly uniform distribution of processor time, as we would with the fair-share scheduler). Light-gray *art* threads get considerably less processor time with the TMR scheduler, because they are the high-PCMR threads.

A common way to measure fairness of processor time distribution is to measure the coefficient of variation (standard deviation expressed as a percentage of the mean) of processor time fractions. With the default scheduler, the coefficient of variation is 43%, and with the target-miss-rate scheduler it is 90%.

Because high-PCMR threads get a smaller share of processor time than with the default scheduler, their individual performance suffers despite the improved overall performance of the system. Individual performance degradation of high-PCMR threads was comparable to the fraction by which their processor time share reduced, as compared to the default scheduler, because the improvements in the individual instructions per cycle (IPC) that they experienced were negligible.

Although the system completes more instructions per cycle, most instructions are executed by the low-PCMR threads. Overall miss-rate improvements produced by the target-miss-rate algorithm are not sufficient to offset the performance degradation of high-PCMR threads due to the reduced number of processor cycles. We provide more analysis and data supporting this conclusion in Section IV.4.5.

#### *IV.4.4.4 Runtime overhead*

We found that the target-miss-rate algorithm imposes no measurable performance overhead. We measured the overhead of the algorithm on a conventional, not chip-multithreaded, UltraSPARC® II server with four processors, each running at 450 MHz. We used a conventional processor, because our goal was to measure only the overhead, and not the effects of scheduling. Using real hardware allowed us to perform longer experiments. Since the target-miss-rate policy has no effect on a conventional processor, we set the target miss rate threshold to 100%. As a result, the TMR scheduler did not schedule threads any differently from the default scheduler, but still performed its measurements and ran its algorithms, generating the associated overhead.

We instrumented the scheduler to perform multiple measurement phases, to simulate the actions of a scheduler enhanced to account for program phase changes. The scheduler repeated the measurement phase every 250 milliseconds.

We ran a workload of twelve SPEC CPU2000 integer benchmarks and observed no measurable performance degradation: the entire workload completed in 63 minutes with the default scheduler as well as with TMR scheduler. The expensive operations of the TMR scheduler, i.e. solving the system of linear equations (400µs on our test system), and MAX_PCMR assignment algorithm (800µs on our test system), are performed infrequently and therefore do not have measurable impact on performance.

#### *IV.4.4.4.1 Duration of the measurement phase*

On our simulated processor with eight thread contexts and 18 threads running SPEC CPU2000 benchmarks, we observed an average duration for the measurement phase of 9.8 seconds and a median value of 11.5 seconds (MIN_RUNLENGTH was set to 2 milliseconds). The measurement phase imposes no performance penalties on applications, because it involves passive observation of cache miss rates. Our target application domain is long-running server applications which may persist for days or weeks, and it is acceptable for the system to take 10-15 seconds to prepare for the scheduling phase. It would also be acceptable for other applications, whose lifetimes can be measured in minutes (e.g., desktop user applications).

Re-measuring PCMRs can be performed concurrently with the scheduling phase, i.e., the scheduling phase does not have to be terminated if the scheduler needs to re-measure PCMRs, so a multi-second duration of the measurement phase is not an issue.

The measurement phase is considerably shorter for threads that enter the system once we have estimated PCMRs for many other threads: estimating the PCMR in that case takes about 0.5 seconds.

## IV.4.5 Applicability of results

In order to improve the overall system miss rate, the target-miss-rate algorithm reduces the processor share of high-PCMR threads. This is a necessary measure to improve the overall system miss rate. We were interested to know whether there are situations when the performance improvements experienced by high-PCMR threads due to reduced system miss rate compensate for the performance degradation experienced by those threads due to reduced processor share. In this section we report results of the study addressing this question.

The extent of the overall miss rate improvement depends on individual contribution to the overall miss rate of high-PCMR threads: the higher the thread's individual contribution the higher the benefit of running that thread less frequently. The extent of performance benefit a thread experiences from a reduced miss rate depends on two factors: (1) a thread's *miss-rate sensitivity* – i.e., how much a thread's individual miss rate is affected by the changes in the overall system miss rate, and (2) a thread's *performance sensitivity* – i.e., how much a thread's performance changes when its miss rate changes. We analyzed miss-rate and performance sensitivities of several benchmarks and applications and concluded that performance boost from the overall miss-rate improvements is not likely to compensate for the reduced processor time for high-PCMR threads. In this section we present the data that led us to this conclusion.

### IV.4.5.1 Experimental configuration

To study this phenomenon we created a special version of the target-miss-rate scheduling module, designed to simplify and improve control of the experiments. We configured the scheduler not to perform the measurement phase and instead begin right away with the scheduling phase. We manually assigned PCMR values to the programs and encoded the PCMR of each program in the name of the executable; the scheduler parsed the name of the executable to determine the PCMR. This allowed us to shorten the experiments and to assure that we always begin the measured run at exactly the same point of the program.

We also changed the way the scheduler changes threads' CPU preferences: instead of using the target miss-rate threshold to affect how much the scheduler restricts the CPU preferences of high-PCMR threads, we introduced a static parameter that controls how many systems' processors are off-limits to high-PCMR threads. We call this parameter the *number of squeezes*, because as this parameter increases, the high-PCMR threads are "squeezed" to progressively fewer processors. For example, on a system with four virtual processors, if the number of squeezes is one, the high-PCMR threads can run on all but one system processor, if the number of squeezes is two, they can run on all but two system's processors, and so on. A higher value for the number of squeezes implies more restrictions.

Note that this experimental configuration changes the meaning of the PCMR: instead of using it to estimate the system's miss rate we use it to distinguish high-PCMR threads from low-PCMR threads, so that the scheduler can restrict the CPU preferences of high-PCMR threads, but not of low-PCMR threads. This is exactly what we wanted to accomplish for this study: to investigate how controlled "squeezing" of high-PCMR threads affects the overall miss rate and performance of threads in the system.

#### *IV.4.5.2 The workload*

For this study we experimented with the SPEC CPU2000 *art* benchmark, the Berkeley DB benchmark (*db)*, and two synthetic benchmarks. We now describe why we chose these benchmarks.

We measured cache miss-rate sensitivities of all SPEC CPU2000 INT benchmarks, the SPEC FP benchmark *art*, SPEC JBB, and Berkeley DB. We chose *art* and *db* because they experience significant performance improvements when system miss rate is reduced. If those highly sensitive benchmarks do not experience significant enough performance benefits to offset the reduced share of processor time, then insensitive benchmarks will not either.

In addition to miss-rate sensitivity, we cared about performance sensitivity, i.e., to what extent the L2 miss rate is a dominant factor in the benchmark's performance. Note that in Section IV.4 we express L2 miss rate in *misses per instruction* (MPI), because comparing threads' misses per instruction allows us to compare their relative performance sensitivities. A thread with a higher L2 MPI will have a higher performance sensitivity, because it will spend more of its runtime handling L2 cache misses. For example, *art*'s performance is more sensitive to its L2 miss rate that *db*'s performance, because *art*'s L2 MPI is several times higher than *db*'s L2 MPI. So by using these benchmarks we also get a good spectrum of performance sensitivities.

In our experiments, we used *art* and *db* as high-PCMR threads: the scheduler reduced the CPU preferences of these threads allowing us to analyze the trade-off between reduced share of processor time and performance improvements.

We also created two synthetic benchmarks to use as low-PCMR threads, *synth1* and *synth2.* These programs traverse an array, referencing some parts of the array more frequently than others. We constructed those benchmarks to have similar miss-rate sensitivity to *art* and *db.*

Figures 4-8 and 4-9 show miss-rate sensitivities of our benchmarks. We show *art* on a different graph, because it has a much higher miss rate than the other benchmarks, and showing the other benchmarks on the same figure as *art* obscures their curves.

### *IV.4.5.3 The experiment*

We use the experimental processor with eight virtual processors described in the beginning of this section to run two workloads, workload #1 and workload #2.

Workload #1 consists of eight threads running *art* (the high-PCMR thread) and eight threads running *synth2* (the low-PCMR thread). For workload #1 we configure the system with a 6MB L2 cache, because the shape of the sensitivity curve for *art* (Figure 4-8) changes depending on the cache size, and we experimentally determined that using a 6MB cache gives us the highest miss-rate sensitivity for *art* when *art* competes

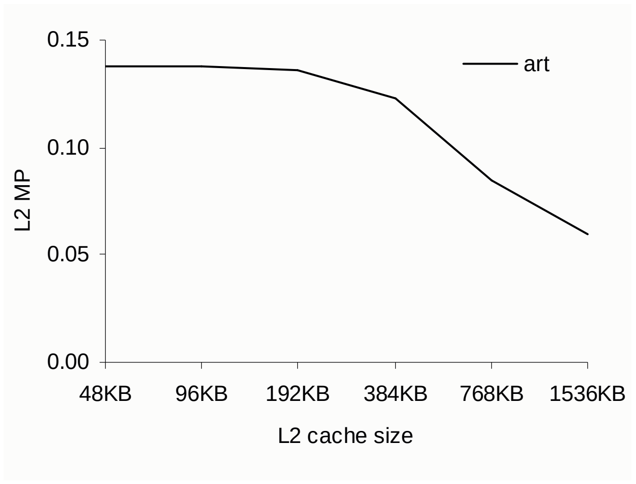


**Figure 4-8. *Art*'s miss-rate sensitivity**
Y-axis shows the number of misses per instruction.

for the cache with other benchmarks.

Workload #2 consists of eight threads running *db* (the high-PCMR threads) and eight threads running *synth1* (the low-PCMR threads). We configure the system's cache size to be 384KB – we experimentally determined that this gives us the highest miss-rate sensitivity for *db.*

We experiment with four settings of the *number of squeezes* parameter: zero, one, two and four.

### *IV.4.5.4 The results*

In this section we use the experiments described above to demonstrate the following: (1) improvement in the overall system miss rate depends on the high-PCMR thread's individual contribution to the overall miss-rate, (2) a thread's performance benefit due to improved system's miss rate depends on its miss-rate sensitivity and performance sensitivity, and (3) performance benefits experienced by high-PCMR

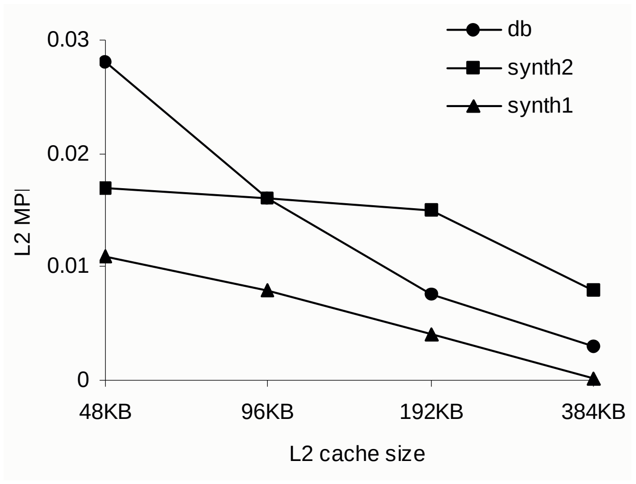


**Figure 4-9. Miss-rate sensitivities of *db, synth1* and *syth2***
Y-axis shows the number of misses per instruction.

threads are generally not large enough to offset the reduction in processor time share necessary to create the overall miss rate improvements.

#### *IV.4.5.4.1 Effect of PCMRs on the overall miss rate*

Figure 4-10 shows normalized average per-thread miss rate improvements for workload #1. Threads running *synth2* experience a 32% reduced L2 miss rate for four squeezes. Threads running *art* experience a 15% reduced L2 miss rate for four squeezes. Figure 4-11 shows the normalized average per-thread miss rate improvement for workload #2. Miss rate improvements are smaller: *db* experiences 18% reduction in its miss rate, *synth1* – no improvement at all.

The reason for smaller L2 miss rate reduction for workload #2 is because *db,* the high-PCMR thread of workoad #2, has a much smaller

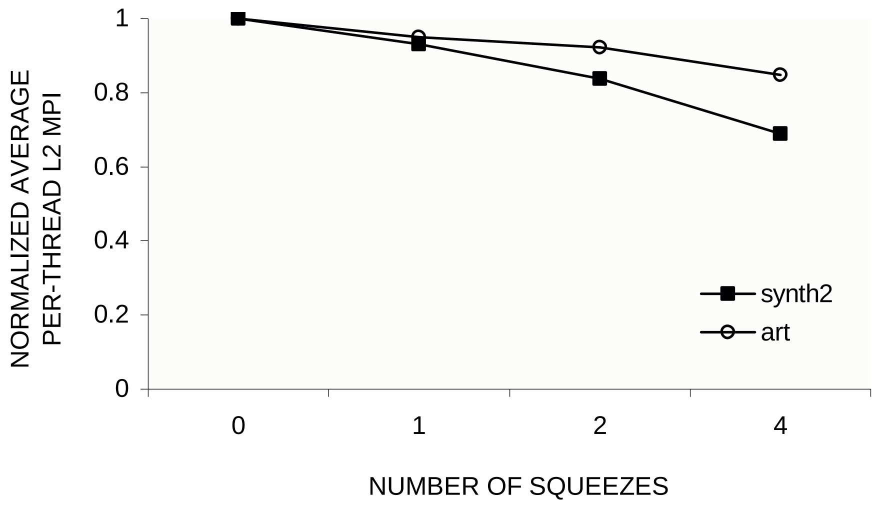


**Figure 4-10. Per-thread miss-rate improvement for workload #1**

individual L2 cache miss rate than *art,* the high-PCMR thread of workload #1. Therefore, in workload #2 “squeezing out” the high-PCMR threads has a smaller effect on the overall system miss rate than for workload #1. For workload #1, the system’s miss rate reduces by 40% (from 0.16 to 0.10 misses per L2 reference) at four squeezes compared with zero squeezes. In contrast, the overall L2 miss rate for workload #2 reduces by 10% (from 0.10 to 0.09 misses per L2 reference) at four squeezes compared to zero squeezes.

### *IV.4.5.4.2 Effect of miss rate reduction on performance*

Figure 4-12 shows the normalized average per-thread IPC for each thread type for workload #1. *Art* experiences IPC improvement of 8% at four squeezes, due to a 15% improvement in its L2 cache miss rate (Figure 4-10). *Synth2* experiences no improvement in the IPC: although

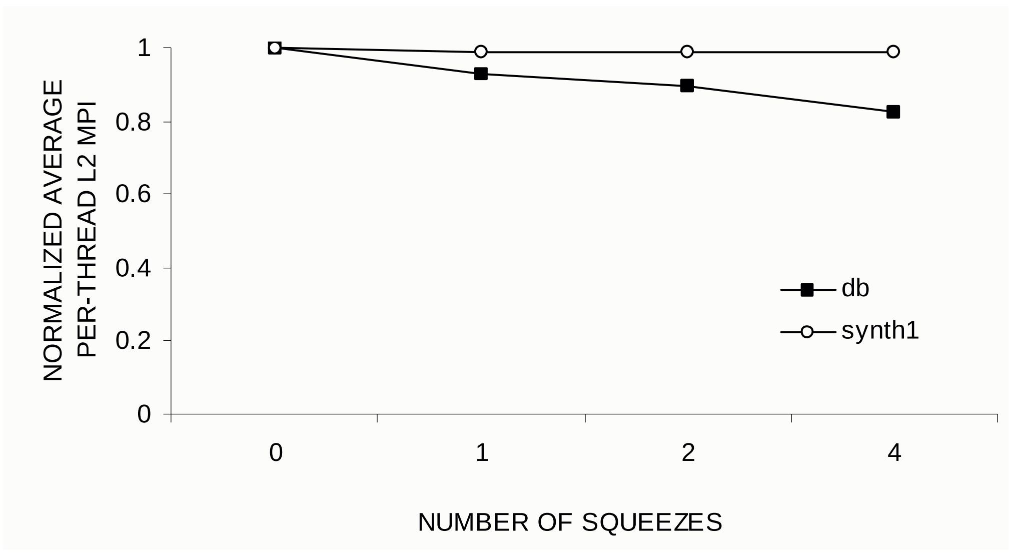


**Figure 4-11. Per-thread miss-rate improvement for workload #2**

its miss rate reduces by 31% at four squeezes, at the rate of 0.004 misses per instruction, its performance is not sensitive to the L2 miss rate.

Figure 4-13 shows normalized average per-thread IPC for workload #2. *Db* experiences a 7% higher IPC at four squeezes than at zero squeezes due to an 18% reduction in its L2 miss rate (Figure 4-11). There is no IPC improvement for *synth1*, because its miss rate remained unchanged (Figure 4-11).

Note that because *art*'s performance is more sensitive to the L2 cache miss rate than *db*'s performance, *art* experienced larger performance benefits: a 15% reduction in the L2 miss rate for the threads running *art* yielded an 8% improvement in the IPC. For *db*, an 18% reduction in the L2 miss rate yielded a lower, 6%, improvement in

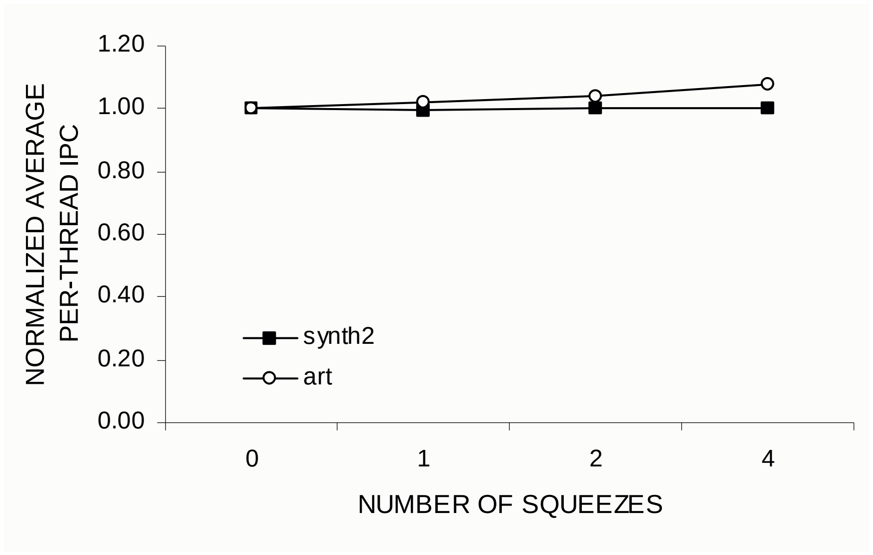


**Figure 4-12. Instruction-per-cycle improvement for workload #1**
Y-axis shows normalized average per-thread instructions per cycle for each thread type

the IPC.

The high-PCMR threads *art* and *db* experienced IPC improvements of 8% and 6%. However, in order to produce these improvements, the processor share of these high-PCMR threads was significantly reduced. At four squeezes, those high-PCMR threads received roughly 38% less processor time than at zero squeezes. The improvements in the IPC were not sufficient to compensate for the reduced processor-time share. As a result, overall throughput of threads running *art* and *db* reduced by about 30 and 32%.

### *IV.4.5.5 Conclusions*

We conclude that given miss-rate and performance sensitivities typical to modern workloads it is unlikely that the IPC improvements that high-PCMR threads experience with the target-miss-rate algorithm

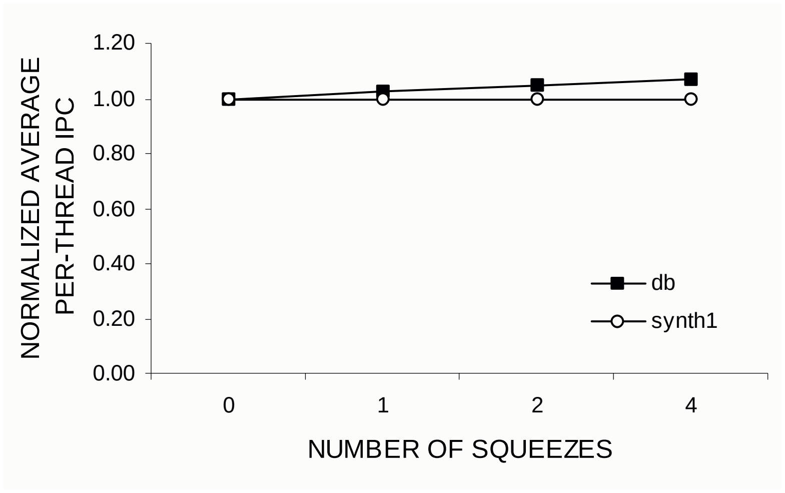


**Figure 4-13. Instruction-per-cycle improvement for workload #2**
Y-axis shows normalized average per-thread instruction per cycle for each thread type.

will compensate for those threads' reduced processor time shares.

Even if memory-system characteristics changed such that a thread's performance would be completely dominated by L2 cache misses, in most cases, threads' miss-rate sensitivities are not high enough to create room for IPC improvements that would offset the reduction in processor time share. A hypothetical workload with sufficiently high miss-rate sensitivity would have memory access patterns "falling-off-a-cliff", i.e., a workload sequentially traversing its working set, such that the miss rate is very poor when the working set does not fit in the cache, but a small change in the number of cache lines allocated for this workload makes the working set fit in the cache and results in a perfect cache hit rate.

## IV.5 Summary and Discussion

We presented the design, implementation and evaluation of the target-miss-rate algorithm. The target-miss-rate algorithm improves system miss rate and throughput by dynamically detecting the threads that contribute most to the overall miss rate and running those threads less frequently. On the downside, the algorithm decreases individual performance of high-miss-rate threads. While improving throughput is a reasonable goal in many domains, excessive focus on throughput may hurt fairness, as we have demonstrated.

Our experience with the target-miss-rate algorithm showed that the system throughput may significantly vary depending on the mix of jobs running on the system, and maintaining a target miss rate helps improve performance predictability. Predictable system performance simplifies the design of soft real-time applications and quality-of-service systems and facilitates performance tuning.

# Chapter V. THE CACHE-FAIR ALGORITHM

Our experience with the target-miss-rate algorithm motivated us to search for a solution that addresses performance predictability but does not have the downsides of the target-miss-rate algorithm. The cache-fair algorithm presented in this chapter improves performance predictability and ensures that there are no negative performance effects on critical user applications.

## V.1 Motivation

The cache-fair scheduling algorithm addressees the effects of co-runner-dependent cache sharing on chip multiprocessors. As we demonstrated in Chapter II, L2 cache is a performance-critical resource, and unpredictable, co-runner-dependent sharing of this resource (Figure

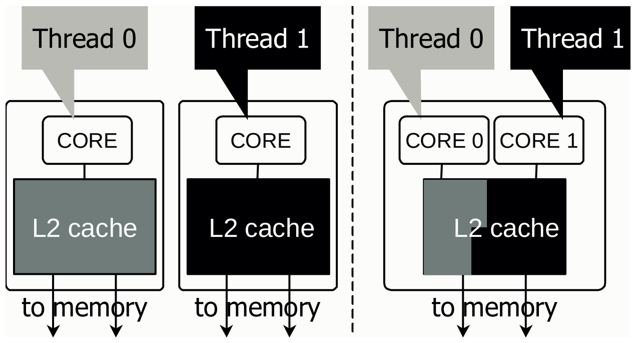


**Figure 5-1. Co-runner-dependent cache-sharing on chip multiprocessors**
Conventional processor (left) and multicore processor (right). Cache coloring shows the thread's working set. Each thread's working set is large enough to populate the entire L2 cache, as shown on the left. However, when the threads become co-runners on a multicore processor (right), the cache is not equally allocated among the threads.

5-1) causes unpredictable, co-runner-dependent performance. Co-runner-dependent performance variability can cause unfair CPU sharing, priority inversion, and inaccurate CPU accounting:

(1) Unfair CPU sharing: Conventional schedulers ensure that equal-priority threads get equal shares of the CPU. On multicore processors a thread's share of the CPU, and thus its forward progress, depends both upon its CPU quantum and the cache behavior of its co-runners. We demonstrated in Chapter II that a SPEC CPU 2000 benchmark *twolf* runs 28% slower with co-runner *art* than with *crafty,* even though *twolf* executes exactly the same set of instructions in both cases. Other studies report similar performance variability [22,39,42].

(2) Priority inversion: A priority-based scheduler on a conventional processor ensures that elevating a job's priority results in greater forward progress for that job. On a multicore processor, if the high-priority job is scheduled with 'bad' co-runners, it will experience inferior rather than superior performance.

(3) Inaccurate CPU accounting: On systems where users are charged for CPU hours [2,7] conventional schedulers ensure that processes are billed proportionally to the amount of computation accomplished by a job. On multicore processors, the amount of computation performed in a CPU

hour varies depending on the co-runners, so charging for CPU hours is not accurate.

To achieve fair sharing on these processors, the operating system must consider L2 cache allocation. This problem is similar to fair sharing in a shared-memory multiprocessor [19], but the solution in the context of chip multiprocessors must be quite different. The difference is that the operating system can control multiprocessor memory allocation, while L2 cache allocation is outside the operating system's control.

Our cache-fair algorithm redistributes CPU time to threads to account for unequal cache sharing: if a thread's performance decreases due to unequal cache sharing it gets more time, and vice versa. The challenge in implementing this algorithm is determining how a thread's performance is affected by unequal cache sharing using limited information from the hardware. Our solution uses runtime statistics and analytical models and does not require new hardware structures or operating system control over cache allocation.

We implemented the cache-fair algorithm as an extension to the Solaris 10 operating system and evaluated it on our simulated UltraSPARC T1 processor. We demonstrate that our algorithm virtually eliminates co-runner-dependent performance variability. The benchmarks that experience performance variability of 5-28% with the default scheduler, experience variability of under 4% with the cache-fair scheduler.

In Section V.2 we describe the structure of the cache-fair algorithm; in Section V.3 we describe the analytical models used in the algorithm; in Section V.4 we describe the implementation; and in Section V.5 we evaluate the algorithm.

## V.2 The Structure of the Algorithm

The cache-fair algorithm reduces co-runner-dependent performance variability by redistributing CPU time such that a thread runs as quickly as it would with an equally shared cache, regardless of its co-runners.

More specifically, it makes the thread's *CPU latency*, i.e., the time to complete a logical unit of work (such as 10 million instructions) equal to its CPU latency under equal cache sharing. A thread's CPU latency is the product of its *cycles per instruction* (CPI) (how efficiently it uses the CPU cycles it's given) and its share of CPU time (how long it runs on the CPU). Co-runner-dependence affects a thread's CPI, because, for example, a cache-starved thread incurs more memory stalls, exhibiting a higher CPI. An OS scheduler, on the other hand, influences the thread's CPU latency by adjusting its share of CPU time.

Figure 5-2(a) illustrates co-runner dependency. There are three threads (A though C) running on a dual-core processor. Thread A is cache-starved when it runs with Thread B and has better performance when it runs with Thread C. In the figure, a box corresponds to each thread. The height of the box indicates the amount of cache allocated to the thread. The width of the box indicates the CPU quantum allocated to

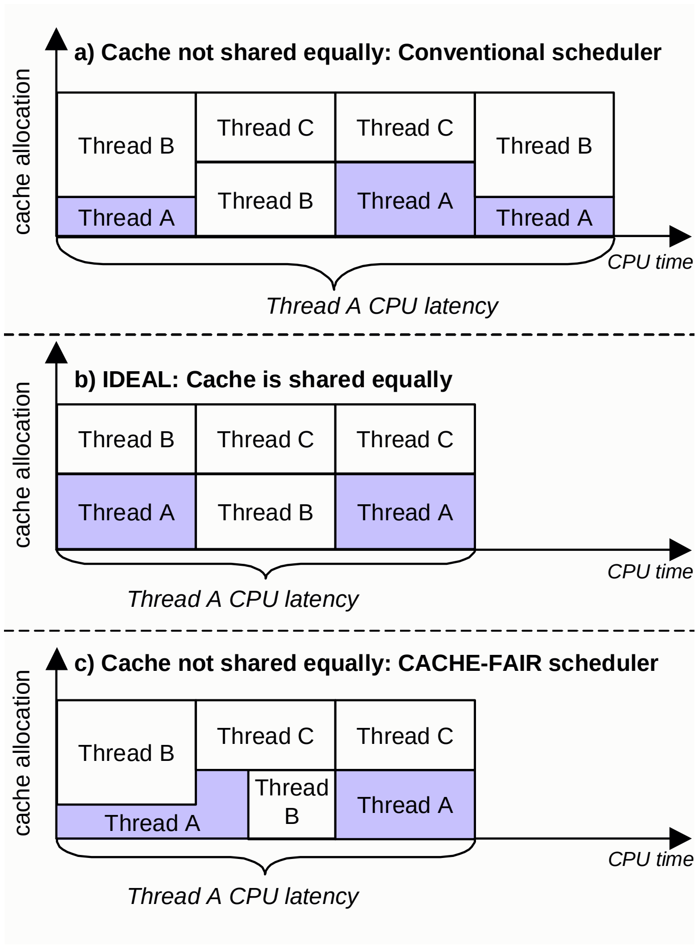


**Figure 5-2. Cache-fair algorithm compensates for unequal cache sharing**
Figure (a) shows thread A's CPU latency when the cache is not shared equally and a conventional scheduler is used, Figure (b) – the ideal scenario when the cache is shared equally and a conventional scheduler is used, and Figure (c) – when the cache is not shared equally and the cache-fair scheduler is used. Thread A takes longer to complete the shaded work in (a) than in (b) or (c).

the thread. The area of the box is proportional to the amount of work completed by the thread. Thread boxes stacked on top of one another indicate co-runners.

Figure 5-2(b) depicts the imaginary ideal scenario: the cache is shared equally, as indicated by the equal heights of all the boxes. Our goal is to complete the same amount of work (the shaded area) in the same time (the length along the X-axis) it takes in the ideal scenario.

In Figure 5-2(a) Thread A completes less work per unit of time than in Figure 5-2(b), because it does not get an equal share of the cache when running with Thread B. As a result, it takes longer to complete the same amount of work; its latency is longer than its latency under equal cache sharing.

Figure 5-2(c) shows how the cache-fair algorithm eliminates the dependency of Thread A's performance on Thread B. By giving Thread A more CPU cycles, it makes Thread A retain the same latency as under equal cache sharing (Figure 5-2(b)). Note that the adjustment to Thread A's CPU quantum is *temporary*. Once the thread catches up with its work, its CPU quantum is restored to its initial value.

Giving more CPU time to one thread takes time away from another thread (or other threads in general), and vice versa. Figure 5-2(c) illustrates that our algorithm gives more time to Thread A at the expense of Thread B. Our algorithm requires, therefore, that there are two classes of threads in the system: the *cache-fair* class and the *best-effort* class. In our example, Thread A is a cache-fair thread, and Thread B is a best-effort thread. The scheduler reduces co-runner-dependent performance variability for threads in the cache-fair class, but not for threads in the best-effort class. Our algorithm, however, avoids imposing significant performance penalties on best-effort threads.

The user specifies the job's class in the same way she specifies a job's priority. We expect that critical user jobs that value performance predictability, such as servers and media-streaming systems, will be in the cache-fair class, and that other, less critical, jobs will fall into the best-effort class. This way we let the user to decide and communicate to the system the importance of each application, allowing the system to appropriately distinguish between critical and non-critical jobs. The system in turn makes sure that critical jobs achieve performance similar to what they would achieve under fair cache sharing, while less critical jobs are subject only to mild performance penalties.

The cache-fair algorithm does not establish a new CPU sharing policy but helps *enforce* existing policies. For example, if the system is using a fair-share policy, the cache-fair algorithm will make the cache-fair threads run as quickly as they would if the cache were shared equally under the fair-share policy.

The key part of our algorithm is correctly computing the adjustment to the thread's CPU quantum. We follow a four-step process to compute the adjustment:

– Determine a thread's *fair L2 cache miss rate* – the number of misses per cycle that the thread would experience under equal cache sharing. We derive the fair L2 miss rate using a new online analytical model (Section V.3.1).

  (We contrast the fair miss with the partial-cache miss rate (PCMR) used in Chapter IV. While the concepts are similar, there are differences. First, the fair miss rate is expressed in misses per cycle, while the PCMR is the percent of references resulting in misses. Second, the fair miss rate is estimated using a different analytical model that the PCMR model.)

– Compute the thread's *fair CPI rate* – the cycles per instruction under the fair cache miss rate. We extend an existing analytical model to perform this computation (Section V.3.2).

– Estimate the *fair number of instructions* – the number of instructions the thread would have completed under the existing scheduling policy if it ran at its fair CPI rate (divide the number of cycles by the fair CPI).

– Measure the actual number of instructions completed.

– Estimate how many CPU cycles to give or take away to compensate for the difference between the actual and the fair number of instructions. Adjust the thread's CPU quantum accordingly.

The algorithm works in two phases:

*Reconnaissance phase:* The scheduler computes the fair L2 cache miss rate for each thread.

*Calibration phase:* The scheduler continuously adjusts, or calibrates, the cache-fair threads' CPU quanta to make them run as they would with an equally shared cache. A single calibration consists of computing the adjustment to the thread's CPU quantum, and then selecting a best-effort thread whose CPU quantum is adjusted to offset the adjustment to the cache-fair thread's quantum. Calibrations are repeated periodically.

New threads in the system are assigned CPU quanta according to the default scheduling policy. For new threads in the cache-fair class, the scheduler performs the reconnaissance phase and then the calibration phase.

The challenge in implementing this algorithm is that in order to correctly compute adjustments to the CPU quanta we need to determine a thread's fair CPI ratio using only limited information from hardware counters. Hardware counters allow measuring threads' runtime statistics, such as the number of cache misses and retired instructions, and, in the case of Intel Core Duo, the number of cache lines allocated to each virtual processor, but they do not tell us how the thread's CPI is affected by unequal cache sharing. We estimate the fair CPI using both information from hardware counters and analytical models.

## V.3 Analytical Models

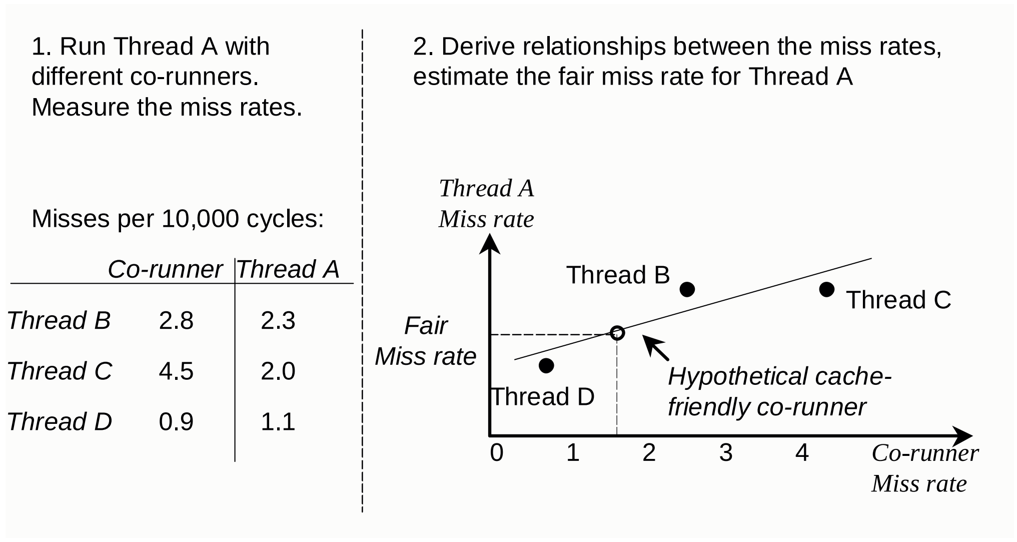
Since our analytical models were designed for use inside an operating system, our goal was to design them such that they impose low runtime overhead and not require any pre-processing of the workload or any *a priori* knowledge about its cache locality. We now present the analytical models used to estimate a) the fair L2 cache miss rate and b) the fair CPI.

### V.3.1 The fair cache miss rate model

As defined above, the fair L2 cache miss rate is the number of *misses per cycle* (MPC) that would be generated by a thread if the cache were shared equally. We need to determine a thread's fair cache miss rate in order to estimate its fair CPI. Modeling cache miss rates is a well-studied area [10,18,21,23,30,42,62,73] but existing models require inputs that are expensive to obtain at runtime. Our model is not a general-purpose cache model, but it produces accurate estimates with low runtime overhead, and thus fits our needs. The fair cache miss rate could also be estimated using our online model for PCMR (Chapter IV), but we do not use that model: For the cache-fair algorithm we were able to design a more accurate online model with stronger theoretical foundation.

Our approach is based on an empirically derived observation that if the co-runners have similar cache miss rates they share the cache roughly equally (we justify this below). So if co-runners A and B experience similar miss rates, they share the cache equally and they each experience their fair miss rate. In this case we say that A and B are *cache-friendly* co-runners. To estimate the fair cache miss rate, for example, for Thread A on a dual-core CPU, one could run Thread A with different co-runners until finding its cache-friendly co-runner. This is not practical, however, because a cache-friendly co-runner may not exist, and if one does exist finding it may require $O(n^2)$ tests. Instead we run Thread A with several different co-runners and derive the relationship between Thread A's miss rate and its co-runners' miss rates. We then use this relationship to estimate the miss rate Thread A would experience with a "hypothetical" cache-friendly co-runner; this miss rate is Thread A's fair miss rate.

Figure 5-3 illustrates this idea. Step 1 shows the miss rates measured as Thread A runs with different co-runners. Step 2 shows that we derive a linear relationship between the miss rate of Thread A and its co-runners. We use the corresponding linear equation to compute Thread


1. Run Thread A with different co-runners. Measure the miss rates.
Misses per 10,000 cycles:
2. Derive relationships between the miss rates, estimate the fair miss rate for Thread A
Thread A Miss rate
Fair Miss rate
Thread B
Thread C
Thread D
Hypothetical cache-friendly co-runner
0
1
2
3
4
Co-runner Miss rate

| | Co-runner | Thread A |
|---|---|---|
| Thread B | 2.8 | 2.3 |
| Thread C | 4.5 | 2.0 |
| Thread D | 0.9 | 1.1 |

**Figure 5-3. Estimating the fair cache miss rate for Thread A.**

A’s miss rate when running with a hypothetical cache-friendly co-runner – its fair miss rate.

Our approach makes the following assumptions: (1) Cache-friendly co-runners have similar cache miss rates, and (2) the relationship between co-runners’ miss rates is linear. We now justify these assumptions.

*Co-runners with similar miss rates share the cache equally:* If we assume that a thread's L2 cache accesses are uniformly distributed in the cache, then we can model cache replacement as a simple case of the *balls and bins problem* [25]. Assume two co-runners, whose cache requests correspond to black and white balls respectively. We toss black and white balls into a bin. Each time a ball enters the bin, another ball is evicted from the bin. If we throw the black and white balls at the same rate, then the number of black balls in the bin after many tosses will form a multinomial distribution centered around one-half. This result generalizes to any number of different colored balls being tossed at the same rate [32]. Thus, two threads with the same L2 cache miss rate (balls being tossed at the same rate) will share the cache equally. We verified this empirically by analyzing how co-runners' cache miss rates correspond to their cache allocations. This theory applies if cache requests are distributed uniformly across the cache. We measured (on our simulator) how cache requests made by SPEC CPU2000 benchmarks are distributed among the cache banks and found that for most benchmarks, the distribution was close to uniform: the difference between the access frequencies of two different cache banks was at most 15%. There were exceptions, *gzip, twolf,* and *vpr,* where such difference reached 50%, but even in those cases our model produced acceptable accuracy, as we will demonstrate shortly.

*The relationship between co-runners' miss rates is linear.* We chose nine benchmarks from the SPEC CPU2000 suite with different cache access patterns and ran them in pairs on our simulated dual-core processor. We ran each benchmark in several pairs. We analyzed the relationship between the miss rate of each benchmark and the miss rates of its co-runners and found that a linear equation approximated these relationships better than other simple functions.

The expression for the relationship between the co-runners' miss rates for a processor with *n+1* cores is:

$$MissRate(T) = a * \sum_{i=1}^{n} MissRate(Ci) + b \qquad \text{(V.1)},$$

where *T* is a thread for which we compute the fair miss rate, *Ci* is the *i*th co-runner, *n* is the number of co-runners, and *a* and *b* are the linear equation coefficients. Thread *T* experiences its fair cache miss rate *FairMissRate(T)* when all concurrent threads experience the same miss rate:

$$FairMissRate(T) = MissRate(T) = MissRate(Ci),$$

for all *i*. Equation (V.1) can be expressed as:

$$FairMissRate(T) = a * n * FairMissRate(T) + b,$$

and the expression for *FairMissRate(T)* is:

$$FairMissRate(T) = \frac{b}{1 - a * n} \qquad \text{(V.2).}$$

The cache-fair scheduler dynamically derives coefficients for Equation V.2 for each cache-fair thread at runtime and then estimates its fair cache miss rate.

We evaluate the accuracy of the fair miss rates estimated using our model by comparing them with the actual fair miss rates. We derived the actual fair miss rate for a thread by measuring the miss rate when this

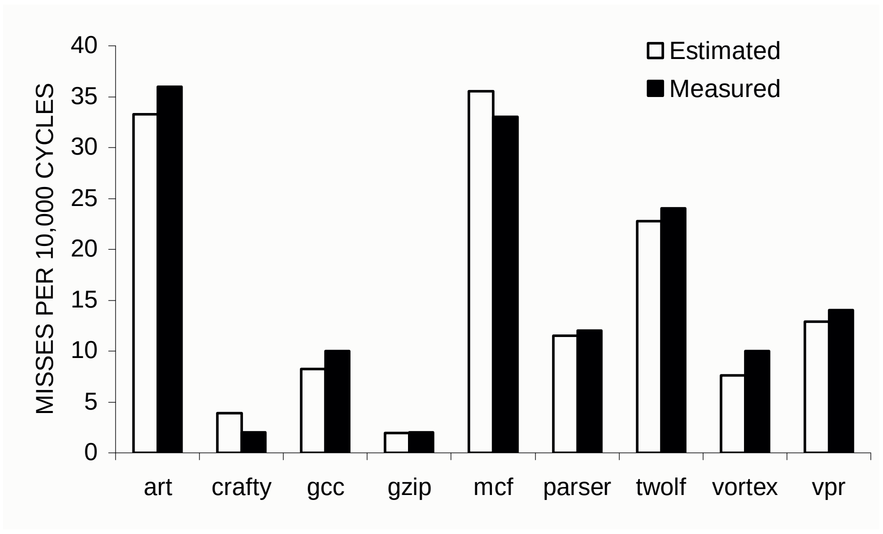


**Figure 5-4. Estimated vs. actual fair cache miss rates**

thread runs with a co-runner on a simulated dual-core machine with an equally partitioned cache (an equally partitioned cache ensures equal cache sharing). We computed the estimated fair miss rate by running each thread with several different co-runners, deriving the coefficients for Equation V.1 by means of linear regression and then using Equation V.2. We validated the model for workloads where there was no data sharing among the co-runners. The effect of data sharing on the model's accuracy is an open question.

Figure 5-4 shows how the estimated fair miss rates compare to the actual miss rates. The names of the SPEC CPU2000 benchmarks are on the X-axis; the Y-axis shows the actual and estimated fair miss rates for each benchmark. The estimated miss rates closely approximate the actual miss rates. The difference between the measured and estimated values is within 8% for six out of nine benchmarks, within 25% for eight out of nine benchmarks. For *crafty*, we overestimated the fair cache miss rate by almost a factor of two, but because *crafty*'s performance is not sensitive to its miss rate (it has one of the smallest miss rates of all benchmarks), a large error does not hurt the algorithm.

In general, we observed that our estimates were less accurate for benchmarks with low miss rates than for benchmarks with higher miss rates. We hypothesize that because low-miss-rate benchmarks actively reuse their working set, there is little variation in the miss rate when running with different co-runners, and because little variation in the data used to build the linear equation, the resulting equation has low fidelity.

Another possible source of errors in the model is the implicit assumption that cache misses occur due to insufficient capacity, rather than changes in the working set or cache line mapping conflicts. To account for these effects one would need to know about the workload's memory access patterns, but this information is expensive to obtain at runtime. Our model provides a good combination of accuracy and efficiency, which makes it practical to use inside an operating system.

A possible limitation of our method is that it requires running the cache-fair thread with multiple different co-runners to estimate its fair miss rate. If the workload has only a few co-runners or if all co-runners have similar cache-access patterns, the linear equation could have a lower fidelity, and our method would be less accurate. A worthy subject of future research would be to investigate the effect of co-runner diversity on the model accuracy, and if the effect is significant to devise an alternative solution.

### V.3.2 The fair CPI model

The fair CPI ratio is the number of cycles per instruction achieved by a thread under equal cache sharing. We use fair CPI to compute the adjustment to cache-fair threads' CPU quanta. To estimate it, we use an analytical model that uses previously described techniques [53,69,76] along with straightforward adaptations for our CPU architecture. The model has the form:

$$CPI = perfectCacheCPI + L2CacheStallsPI \qquad (V.3),$$

where *perfectCacheCPI* is the CPI when there are no L2 cache misses, and *L2CacheStallsPI* is the per-instruction stall cycles due to handling L2 cache misses. The expression for *L2CacheStallsPI* is:

$$\begin{aligned} L2CacheStallsPI = {} & (L2MPI + L2CopybacksPI) * \\ & (MEM\_LATENCY + MembusDelayPerTxn) + \\ & StoreBufferStallsPI \end{aligned} \qquad (V.4),$$

| | |
|---|---|
| *L2MPI* – | the number of cache misses per instruction |
| *L2CopybacksPI* – | the number of transactions (per instruction) writing dirty cache lines back to memory |

| | |
|---|---|
| *MEM_LATENCY* – | memory access time required to fetch a cache line |
| *MembusDelayPerTxn* – | memory-bus delay per memory transaction |
| *StoreBufferStallsPI* – | the number of cycles per instruction a processor stalls on a store buffer |

The expression for fair CPI is:

$$fairCPI = perfectCacheCPI + fairL2CacheStallsPI \qquad (V.5),$$

where $fairCPI$ is the CPI when the thread experiences its fair cache miss rate. To estimate *fairCPI,* we need to determine (1) *perfectCacheCPI* and (2) $fairL2CacheStallsPI$*:*

*Computing perfectCacheCPI:* We compute it using Equation V.3. We measure all the values needed to compute *L2CacheStallsPI* at runtime using hardware performance counters. Subtracting *L2CacheStallsPI* from *CPI* (also measured) gives us the *perfectCacheCPI.*

We validated this CPI model by measuring a benchmark's *perfectCacheCPI* and comparing it with the *perfectCacheCPI* estimated using the model. We measured the *perfectCacheCPI* by simulating the benchmark on our simulator configured with a zero-latency memory system and an unlimited-bandwidth memory bus. We estimated the *perfectCacheCPI* for a benchmark by substituting into our model the values for *L2MPI, L2CopybacksPI, MembusDelayPerTxn,* and

*StoreBufferStallsPI* measured on our simulator configured with a memory latency of 200 cycles, and a memory-bus with 5.2GB/s bandwidth. Table 5-1 shows that the estimated *perfectCacheCPI* is within 1% of the measured *perfectCacheCPI* for all benchmarks.

*Computing FairL2CacheStallsPI:* To compute *FairL2CacheStallsPI* we substitute into Equation V.4 the fair L2 cache miss rate estimated using the fair miss rate model (we express the fair miss rate in misses per instruction, not misses per cycle as in the original definition, to suit the definition of our model). We obtain the values for *L2CopybacksPI, MembusDelayPerTxn,* and *StoreBufferStallsPI* at the same time when we estimate the fair miss rate. We measure the *L2CopybacksPI, MembusDelayPerTxn,* and *StoreBufferStallsPI* for each observed value of

**Table 5-1. Measured vs. estimated *perfectCacheCPI***

| BENCHMARK | perfectCacheCPI | | DIFFERENCE OF ESTIMATED FROM MEASURED |
|---|---|---|---|
| | MEASURED | ESTIMATED | |
| *art* | 5.03 | 4.98 | 0.99% |
| *crafty* | 1.54 | 1.54 | 0.00% |
| *gcc* | 1.73 | 1.72 | 0.58% |
| *gzip* | 1.32 | 1.33 | 0.76% |
| *mcf* | 2.00 | 1.98 | 1.00% |
| *parser* | 1.55 | 1.56 | 0.65% |
| *twolf* | 1.74 | 1.73 | 0.57% |
| *vortex* | 1.84 | 1.85 | 0.54% |
| *vpr* | 1.45 | 1.45 | 0.00% |

the thread's miss rate and derive linear relationships between these statistics and the miss rate:

$$L2CopybacksPI = a0 * L2MPI + b0 \tag{V.6}$$

$$StoreBufferStallsPI = a1 * L2MPI + b1 \tag{V.7}$$

$$MembusDelayPerTxn = a2 * L2MPI + b2 \tag{V.8}$$

We then estimate the values of the *L2CopybacksPI*, *MembusDelayPerTxn*, and *StoreBufferStallsPI* occurring when the thread experiences its fair miss rate by substituting the estimated fair miss rate for *L2MPI* into Equations V.6-V.8.

## V.4 Implementation

We implemented the cache-fair scheduler in the Solaris 10 operating system on top of the fixed-priority scheduling policy: this means that the scheduler ensures that each cache-fair thread runs as quickly as it would with an equally shared cache under the fixed-priority policy.

As described in Section V.2, each cache-fair thread goes through two phases, the reconnaissance phase and the calibration phase. Figure 5-5 summarizes the phases. In this section we describe their implementation.

### V.4.1 Reconnaissance phase

During the reconnaissance phase, the cache-fair algorithm estimates a thread's fair cache miss rate (Section V.3.1) and generates values for *L2CopybacksPI, MembusDelayPerTxn,* and *StoreBufferStallsPI* used to compute *fairCPI* (Section V.3.2).

Assume that a cache-fair thread *T* is in its reconnaissance phase. As *T* appears on a processor concurrently with a group of co-runners, the scheduler measures the statistics used to estimate *T*'s fair miss rate and its fair CPI. We do not force *T* to run with specific co-runners, but observe any co-runner combinations that appear on the processor. We ensure that *T* executes at least 10 million instructions in each co-runner group. This helps us get a representative measurement. By the end of the reconnaissance phase, we have ten such data points.

We use these data to generate the equations for the fair-miss-rate

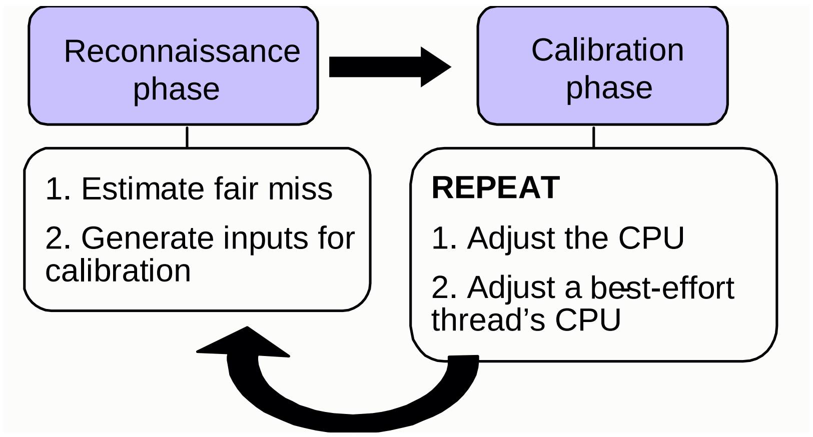


**Figure 5-5. The phases of the cache-fair scheduling algorithm**

and fair-CPI models described in Section V.3. Generating equations requires linear regression analysis, which is usually implemented using floating-point instructions. Because floating-point instructions are not permitted in the Solaris kernel, we implemented linear regression using only integer types.

To collect our measurements, we use hardware counters commonly available on modern processors [5,8,37]. The UltraSPARC T1 processor, which we use as the model for our simulator, does not have a counter for the number of write-back transactions (needed to compute *L2CopybacksPI*). Without this counter, our *fairCPI* model had poor accuracy. We implemented this counter in our simulator, reasoning that since other popular processors, such as those from Intel's Pentium family [37], provide such a counter, it is reasonable to expect that this counter will exist on future processors[7].

## V.4.2 Calibration phase

After the reconnaissance phase, thread *T* enters the calibration phase, where the scheduler periodically adjusts, or calibrates, *T*'s CPU quantum.

[7] We also needed to measure separately the write-back transactions triggered by read and instruction-fetch misses and the write-back transactions triggered by write misses. The stall time on write-triggered write-back transactions is accounted for by the store buffer stall cycles, so we do not need to account for them. We implemented a separate counter for read-triggered write-back transactions. If such separate counter is unavailable on an actual processor, the number of read-triggered write-back transactions can be accurately modeled using the write miss rate and the number of total write-back transactions. We present this model in Appendix IX.2.

The scheduler measures the actual CPI using hardware counters and estimates the fair CPI as described in Section V.3.2. The scheduler compares *T*'s fair CPI to its actual CPI and estimates the temporary adjustment to *T*'s CPU quantum, so that by the end of the next quantum, *T* completes the number of instructions corresponding to its fair CPI. If *T* has been running faster than it would at its fair CPI, the scheduler decreases is CPU quantum. If *T* has been running slower, the scheduler increases its CPU quantum.

The scheduler also selects a best-effort thread whose CPU quantum is adjusted to offset the adjustment to *T*'s CPU quantum. If the scheduler increases *T*'s CPU quantum it decreases that best-effort thread's quantum by the same amount, and vice versa.

If we need to decrease a best-effort thread's CPU quantum, we always select the thread that experienced the smallest penalty to its CPU quantum so far. If we need to increase a best-effort thread's CPU quantum, we select the thread that has suffered the largest penalty. The adjustment to the best-effort thread's CPU quantum is temporary: once the thread has run with the adjusted quantum, its quantum is reset to its original value.

### V.4.3 Phase duration and repetition

The reconnaissance phase needs to be sufficiently long to capture the characteristic cache access patterns of the workload. Upon analysis of temporal variation of the L2 cache access patterns for the nine SPEC CPU2000 benchmarks we use for experiments in this section (Section V.5.1), we found that most workloads (eight out of nine) had largely unchanging caching behavior over time, and though there was occasional short-term variability, any window of 100 million instructions captured the long-term properties of the workload (see Appendix IX.3). Accordingly, we set the duration of the reconnaissance phase to 100 million instructions.

In the calibration phase, we adjust a thread's CPU quantum every fifty million instructions. Frequent adjustments allow the cache-fair algorithm to be more responsive.

To account for program phase changes, the scheduler could repeat a reconnaissance phase at a fixed interval or, ideally, detect when a thread changes its caching behavior and repeat the reconnaissance phase at that point. While dynamic online phase-detection algorithms exist for conventional single-threaded processors [16,64], developing such an algorithm for chip multithreaded processors is an open problem.

## V.5 Evaluation

We evaluate the cache-fair algorithm's ability to reduce co-runner-dependent performance variability, its effect on the overall system throughput and individual thread performance, and its runtime overhead. We also compare our solution with dynamic cache partitioning

– a simple hardware mechanism allowing to partition the cache among co-runners for the sake of fairness or performance optimization [42,73]. We begin by introducing our experimental setup and the workload.

## V.5.1 Experimental configuration

We configure our simulated machine with two single-threaded cores and a shared L2 cache of 1MB. The cache-size and the number of cores were chosen to match Core Duo – a recent dual-core processor from Intel [3].

Our workload consists of the following SPEC CPU2000 benchmarks: *art, crafty, gcc, gzip, mcf, parser, twolf, vortex, vpr.*

## V.5.2 The experiment

The purpose of this experiment is to evaluate (1) improvements in performance predictability with the cache-fair scheduler, (2) the algorithm's effect on the overall system throughput and on performance of individual threads, (3) the algorithm's worst-case performance effect on best-effort threads, and (4) the algorithm's performance overhead.

We run each benchmark in two scenarios *slow schedule* and *fast schedule.* In the slow schedule, the benchmark's co-runners consist mainly of high-miss-rate threads, and in the fast schedule case of low-miss-rate threads. In the slow schedule, the benchmark is likely to run more slowly than in the fast schedule, hence the naming of the two cases. Table 5-2 shows the schedules for each benchmark.

In each of the schedules, the *principal* benchmark belongs to the cache-fair class and one of its three co-runners (randomly selected) is in the best-effort class. Note that because there is only one best-effort thread, we are unable to balance CPU quantum penalties among multiple best-effort threads, allowing us to gauge the worst-case performance effect on best-effort threads.

We constructed this experiment such that the principal benchmark runs with three identical co-runners in each schedule. This gave us the controlled environment where we could be certain that any benefits with the cache-fair scheduler are due to our scheduling algorithm, and not to any co-scheduling effects. Because the benchmark runs with identical co-runners it would not make sense to estimate its fair miss rate in this environment: we would not have enough different data points for the linear regression. Therefore, we estimate all benchmarks' fair miss rates in a separate experiment, where all principal benchmarks run together, and then use that estimated fair miss rate in the experiment.

We run each schedule until the principal benchmark completes one-half billion instructions in the calibration phase; running to completion would take weeks on a simulator. We fast-forward the simulation to the point where all benchmarks enter the main processing loop and then perform the detailed simulation. All benchmarks are CPU-bound.

**Table 5-2. The schedules for each benchmark**

| *Principal* | *Fast Schedule* | *Slow Schedule* |
|---|---|---|
| *art* | *art,crafty,crafty,crafty* | *art,mcf,mcf,mcf* |
| *crafty* | *crafty,vpr,vpr,vpr* | *crafty,mcf,mcf,mcf* |
| *gcc* | *gcc,vpr,vpr,vpr* | *gcc,mcf,mcf,mcf* |
| *gzip* | *gzip,crafty,crafty,crafty* | *gzip,mcf,mcf,mcf* |
| *mcf* | *mcf,gzip,gzip,gzip* | *mcf,crafty,crafty,crafty* |
| *parser* | *parser,crafty,crafty,crafty* | *parser,mcf,mcf,mcf* |
| *twolf* | *twolf,crafty,crafty,crafty* | *twolf,mcf,mcf,mcf* |
| *vortex* | *vortex,crafty,crafty,crafty* | *vortex,mcf,mcf,mcf* |
| *vpr* | *vpr,crafty,crafty,crafty* | *vpr,mcf,mcf,mcf* |

We compare performance with the cache-fair scheduler and with the Solaris fixed-priority scheduler, to which we refer as the *default* scheduler.

## V.5.3 Co-runner-dependent performance variability

We compare the time that it takes the principal benchmark to complete its work segment in the fast and slow schedules. We refer to this quantity as *completion time.* We define *performance variability* as the percent slowdown of the benchmark's completion time in the slow schedule compared to the fast schedule. For example, *twolf*'s performance variability with the default scheduler is 28%, indicating that *twolf* takes 28% longer to complete in the slow schedule than in the fast schedule with the default scheduler.

Figure 5-6 shows the performance variability for our benchmarks with the default scheduler and the cache-fair scheduler. Performance

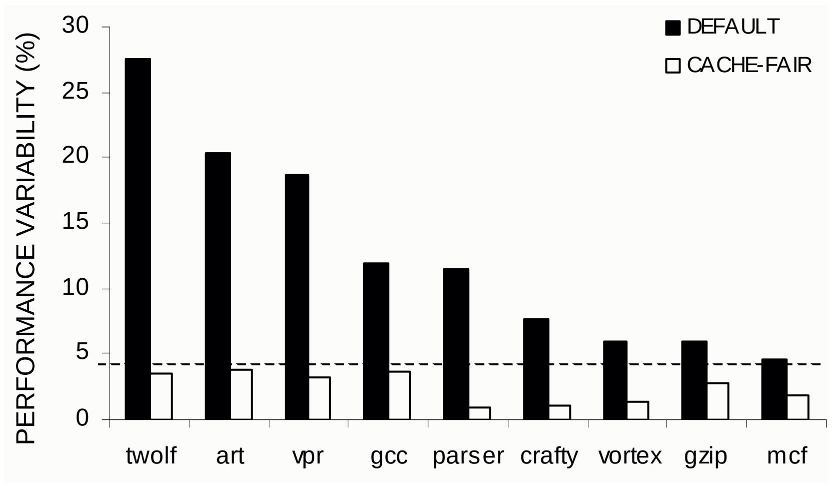


**Figure 5-6. Performance variability with default and cache-fair scheduler**

variability with the default scheduler (black bars) is substantial, ranging from 5%(*mcf*) to 28% (*twolf*), and ranging between 6 and 20% for the rest of the benchmarks.

With the cache-fair scheduler (white bars), performance variability is negligible: in all cases it is below 4%. (The dotted line on the figure indicates the 4% mark.)

This result demonstrates that the cache-fair scheduler was able to accurately compute the adjustments of CPU time quanta that compensated for the difference between the benchmark's actual and fair CPI. This was possible due to the accuracy of our CPI model. For example, when *vpr* runs with the default scheduler, its CPI is 19% higher in the slow schedule than in the fast schedule, resulting in 19% longer completion time in the slow schedule. When *vpr* runs with the cache-fair scheduler, its CPI is still about 18% higher in the slow schedule, but because the cache-fair scheduler compensates for this, its completion time is only 3.81% longer in the slow schedule.

## V.5.4 Effect on throughput and performance

In this section we evaluate the algorithm's effect on overall system throughput and on performance of individual benchmarks.

### *V.5.4.1 Effect on system throughput*

We now evaluate the effect of the cache-fair algorithm on system throughput: i.e., we compare the total number of instructions per cycle that the workload completes with the default scheduler and the cache-fair scheduler.

Figure 5-7 shows the throughput (instructions per cycles) for slow schedules. Black bars show the default scheduler, white bars show the cache-fair scheduler. The x-axis shows the names of the principal benchmarks for each schedule. The numbers for each schedule are normalized to the IPC with the default scheduler.

The data show that with the cache-fair scheduler the overall system throughput generally increases (by 1-13% for 5 out of 9 schedules) or remains unchanged (for 3 out of 9 schedules). The cache-fair scheduler increases the CPU share for the low-miss-rate threads, because those threads are likely to get less than their fair share of cache. Low-miss-rate threads are also more likely to have high individual IPC.

As a result, the system executes more instructions from the low-miss-rate high-IPC threads, increasing the overall throughput.

The only instance where the system throughput decreased (by 12%) was the case where *parser* was a principal benchmark. *Parser*'s L2 miss rate was lower than its estimated fair miss rate, so the cache-fair scheduler reduced *parser*'s CPU share, causing *parser*'s co-runners *mcf* (high-miss-rate low-IPC threads) to run more frequently than with the default scheduler. Further investigation revealed that *parser* co-scheduled with *mcf* uses roughly half the cache, suggesting that in the SLOW schedule, with *mcf* threads as co-runners, *parser* attains its fair miss rate and the scheduler should not reduce *parser*'s timeslice. The fact that the scheduler reduced *parser*'s timeslice indicates that *parser*'s fair miss rate was estimated incorrectly: The estimated fair miss rate was

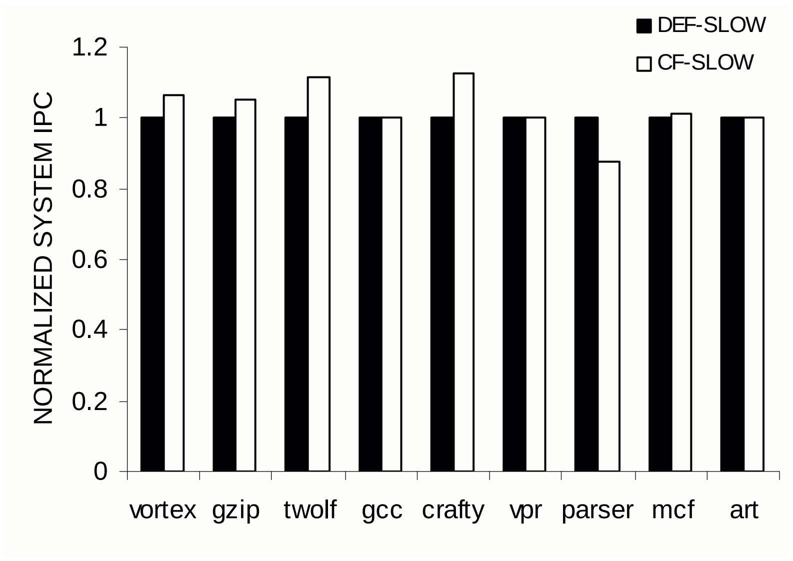


**Figure 5-7. System IPC for slow schedules with the default scheduler and cache-fair scheduler**

higher than the actual fair miss rate. Improving the accuracy of our model for fair miss rates will allow us to avoid this incorrect behavior in the scheduler. A possible way to improve the model is to use the number of cache lines allocated to a thread [37] as one of the model's parameters.

Figure 5-8 compares the normalized throughput for fast schedules. Throughput largely remains unchanged. For 8 out of 9 schedules, the throughput changes by at most +/- 3% in comparison with the default scheduler. In the schedule where *art* is the principal benchmark, the throughput increased by 8%. *Art* is by far the slowest application among all our benchmarks, and with an extremely poor cache locality it usually occupies more than its fair share of the cache. As a result, the cache-fair scheduler reduced *art*'s share of CPU cycles, executing fewer instructions from this low-IPC thread, and increasing the overall throughput.

We conclude that the cache-fair algorithm has largely positive effect on throughput in cases where the workload consists of many high-miss-rate threads. In cases where the workload consists of low-miss-rate threads, the algorithm has a neutral effect on throughput. This demonstrates that improving fairness has in general a positive effect on throughput – a result reported earlier by Kim et al, who proposed a hardware scheme for fair cache sharing [42].

### *V.5.4.2 Effect on individual thread performance*

We now evaluate the effect of the cache-fair algorithm on absolute performance of cache-fair threads. We have demonstrated that the cache-fair algorithm improves performance predictability. But the way it achieves this is by adjusting a thread's CPU share so that it runs as quickly as it would with an equally shared cache. This means that threads that use more than their fair share of the cache will get a smaller share of CPU time than with the default scheduler, and as a result will run slower than with the default scheduler. Analyzing this effect is helps us understand the trade-off between predictability and performance.

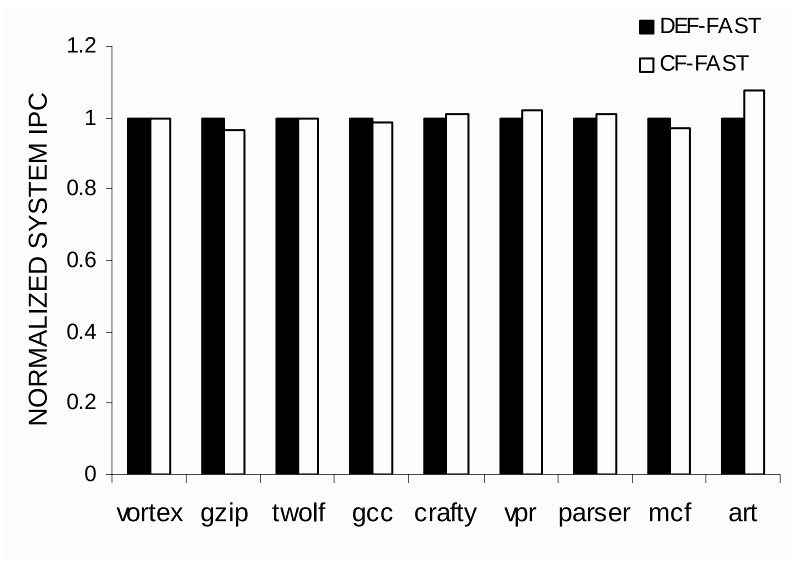


**Figure 5-8. System IPC for fast schedules with the default scheduler and cache-fair scheduler**

Figure 5-9 compares absolute completion times of the principal benchmarks with the default and cache-fair schedulers. The white dots indicate the principal benchmark's completion times with the default scheduler. The higher dot indicates the slow schedule (longer completion time), the lower – the fast schedule (shorter completion time). Black boxes show the completion times with the cache-fair scheduler. The lower side of the box shows the completion time in the fast schedule, the higher side shows the completion time the slow schedule. The numbers are normalized to the completion time in the fast schedule with the default scheduler. Lower position of dots and boxes indicate shorter completion times (i.e., better performance).

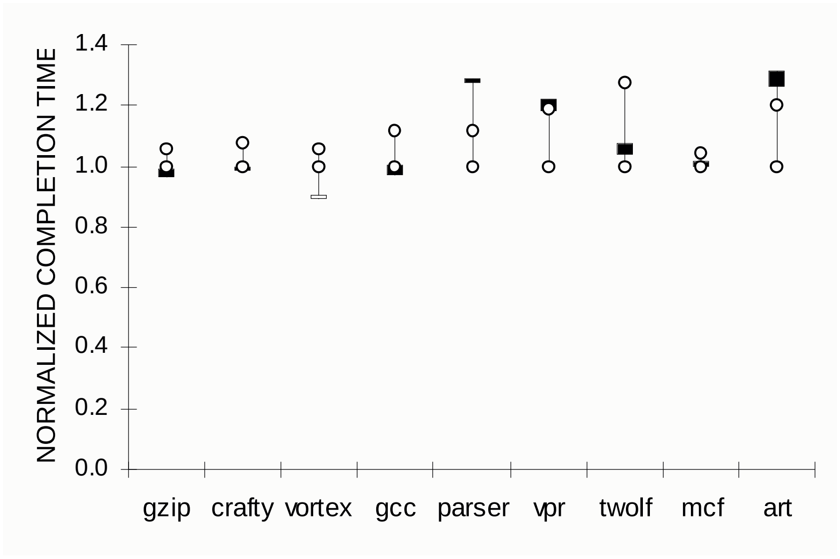


**Figure 5-9. Effect on absolute performance of cache-fair threads**
White dots indicate the principal benchmark's completion times with the default scheduler. The higher dot shows the slow schedule, the lower shows the fast schedule. Black boxes show the completion times with the cache-fair scheduler. The lower side of the box shows the fast schedule, the higher shows the slow schedule. All numbers are normalized to the completion time in the fast schedule with the default scheduler.

In this figure, we sort the applications by their individual performance (using the instructions per cycle (IPC) as the metric for performance). So the applications with high individual IPC appear to the left of the figure, and those with low individual IPC to the right. Note that high-IPC threads usually experience better individual performance with the cache-fair scheduler than with the default scheduler (as indicated by a lower position of the black boxes in relation to white dots). This is what we expect: high-IPC threads are usually also the low-miss-rate threads, who get less than their fair share of the cache with the default scheduler, and the cache-fair scheduler increases their share of CPU time, making their completion times shorter.

### V.5.5 Effect on best-effort threads

We now evaluate the cache-fair scheduler's performance effect on best-effort threads. The cache-fair scheduler adjusts best-effort threads' CPU time quanta in response to the adjustments it makes to cache-fair threads' time quanta. If it gives more CPU time to the cache-fair thread, it takes time away from the best-effort thread, and vice versa. Recall that we designed our experiment such that there was only one best-effort thread in each schedule. Therefore, the CPU share effects were not distributed among multiple best-effort threads like they would if there were multiple best-effort threads in the system. This experimental design

**Table 5-3. Performance degradation for the best-effort thread in each schedule compared with the default scheduler**
Positive numbers indicate performance degradation, negative numbers indicate performance boost.

| SCHEDULE | VICTIM SLOWDOWN (%) | |
|---|---|---|
| *vortex-slow* | | 8.51% |
| *vortex-fast* | | 5.84% |
| *gzip-slow* | | 7.14% |
| *gzip-fast* | | 1.40% |
| *twolf-slow* | | 15.79% |
| *twolf-fast* | | -3.11% |
| *gcc-slow* | | **26.70%** |
| *gcc-fast* | | 11.23% |
| *crafty-slow* | | 4.58% |
| *crafty-fast* | | 2.69% |
| *vpr-slow* | | 2.32% |
| *vpr-fast* | | -13.43% |
| *parser-slow* | | -11.34% |
| *parser-fast* | | **-23.84%** |
| *mcf-slow* | | -0.63% |
| *mcf-fast* | | 6.90% |
| *art-slow* | | -8.07% |
| *art-fast* | | -23.38% |
| | **ARITHMETIC MEAN:** | **0.52%** |
| | **MEDIAN:** | **2.50%** |
| | **MAXIMUM:** | **26.70%** |
| | **MINIMUM:** | **-23.84%** |

allowed us to evaluate the worst-case performance effect on best-effort threads.

Table 5-3 shows performance degradation experienced by the best-effort thread in each schedule. The left column shows the schedules, identified by the principal benchmark, and the type of the schedule. The right column shows how much less CPU time (in percent) the best-effort thread received with the cache-fair scheduler than with the default scheduler. Positive values indicate that the best-effort thread's CPU share was reduced (negative performance effect), negative values indicate that it was increased (positive performance effect).

In general, the cache-fair scheduling algorithm has only a mild performance effect on best-effort threads. The mean reduction to the best-effort thread's CPU share was less than 1%, and the median reduction was 2.5%. In about one third of the schedules, the best-effort threads got a larger CPU share with the cache-fair scheduler than with the default scheduler.

In one case, a best-effort thread was significantly penalized: in the slow schedule where *gcc* was the principal benchmark, the best-effort thread's CPU share was reduced by 27%. This indicates the importance of having multiple best-effort threads and distributing the penalty among them. It is also possible to put a cap on how much the scheduler can reduce a best-effort thread's CPU share.

We believe that the performance effect on best-effort threads could be even less significant if there were multiple cache-fair threads in the system. CPU quantum boost for low-miss-rate cache-fair threads could be offset by the CPU quantum reduction for high-miss-rate cache-fair threads. In this case, the system would rely less on the presence of best-effort threads. Evaluating the system with multiple cache-fair threads is a worthy avenue for future investigation.

### V.5.6 Runtime overhead and scalability

The cache-fair scheduling algorithm was designed and implemented to have low performance overhead. Most of the work done during the reconnaissance phase involves accessing hardware performance counters, requiring only a few processor cycles per access. To estimate the fair cache miss rate, we require only about ten data points for each cache-fair thread, keeping the memory overhead low. A small number of data points also limits the overhead of the linear regression, which runs in *O(N)* steps, where *N* is the number of data points. The calibration phase involves simple arithmetic operations and inexpensive access to hardware performance counters.

We measured the performance overhead of the cache-fair scheduler (i.e. by how much it increases the completion time for each benchmark), and observed that it was under 1% in all cases.

Scalability is an important design goal because the number of cores on multicore processors is likely to increase in the future, and it is important that our algorithm work efficiently on future processor models. Although we have not evaluated the scalability of our algorithm, we believe our design will scale because we avoid inter-processor communication and because the amount of work done by the algorithm is independent of the number of cores.

### V.5.7 Comparison with cache partitioning

Recent research proposed using dynamic cache partitioning on chip multithreaded processors to address unequal cache sharing [22,29,42,60,73]. Dynamic cache partitioning is one of simpler hardware structures designed to address the problems we addressed with the cache-fair algorithm, so it is worth comparing our algorithm with dynamic cache partitioning. Although one study has shown that dynamic cache partitioning can alleviate the effects of “bad” co-runners [42], we found that it does not reduce the co-runner-dependent performance variability to the extent that the cache-fair algorithm does.

We implemented dynamic cache partitioning in our simulated processor in a similar fashion as was done in the recent work [42], and ran our nine principal benchmarks in the slow and fast schedules (shown in Table 5-2) on a dual-core machine with the cache equally partitioned among the cores. Partitioning reduced co-runner-dependent performance variability only for three out of nine benchmarks and made no difference for the remaining six benchmarks. While performance variability due to cache sharing was eliminated, there remained performance variability due to contention for the memory bus. (We confirmed that memory-bus contention was the problem by showing that running with unlimited memory-bus bandwidth eliminated performance variability.) The cache-fair algorithm does not have this problem, because it incorporates memory-bus delay into the fair CPI model, accounting for performance variability due to sharing of the memory bus.

This experiment demonstrated that the problem of performance predictability and unfair cache sharing cannot be addressed with a simple hardware solution. Using a software solution can be preferable to building complex hardware that fully addresses this problem, because a software solution has a lower development cost, greater flexibility, and a shorter time-to-market.

## V.6 Summary and Discussion

We have demonstrated that the cache-fair algorithm virtually eliminates co-runner-dependent performance variability, and does so without special hardware support, and without measurable performance overhead. Co-runner-dependent performance is the result of unequal resource sharing, and by eliminating it, we address the problems caused by unequal resource sharing:

(1) <u>Unfair CPU sharing</u>: With our algorithm, an application achieves predictable forward progress regardless of its co-runners. The effects of unfair resource sharing are negated.

(2) Priority inversion: Our algorithm ensures that a thread makes predictable forward progress regardless of its co-runner. Therefore, elevating a thread's priority results in greater forward progress, and vice versa, just like on conventional processors.

(3) Inaccurate CPU accounting: Our algorithm reduces dependency of a thread's performance on its co-runner. Charging for CPU hours is appropriate, because the amount of work accomplished by a thread is proportional to the CPU hours paid for by the customer and is not affected by the co-runner.

The cache-fair algorithm redefines fairness on chip multithreaded processors. A conventional definition of fairness is fair sharing of CPU cycles. On chip multithreaded processors, where applications share a performance critical resource (the L2 cache), CPU-sharing policies must account for this resource. The cache-fair algorithm incorporates L2 cache sharing into its notion of fairness, distributing CPU cycles to account for unfair sharing.

We chose to accomplish fairness by redistributing CPU cycles. There are other ways to look at fairness. Perhaps the right way to accomplish fairness is by enforcing it in hardware – i.e. dynamically partitioning all shared resources in a way that answers a system's CPU-sharing policy [42,60,73]. While this remains an open question, we believe that a software solution has advantages.

One advantage is shorter time-to-market. While hardware solutions take years to reach a consumer, a software solution can be delivered more quickly. Our cache-fair algorithm runs on chip multithreaded processors that exist today.

Another advantage of our solution is not requiring changes to the hardware/software interface. Changing system interfaces is always precarious. Systems maintain interfaces across models and generations to provide backward compatibility, and if the interface is changed in the "wrong" way, future models will deal with consequences for years. The cache-fair algorithm does not have this problem and thus has a merit as a long-term solution.

# Chapter VI. RELATED WORK

There are two distinct bodies of research relevant to this dissertation. The first is *scheduling for chip multithreaded processors*, including algorithms for improved performance and fairness, as well as those requiring new hardware structures. We discuss them in Section VI.1. The second is *analytical modeling* of caches and CPU performance. This work includes analytical models for chip multithreaded processors as well as models for single-threaded processors relevant to our work. We discuss them in Section VI.2. We summarize in Section VI.3.

## VI.1 Scheduling for Chip Multithreaded Processors

The scheduling algorithms presented in this dissertation pursued three primary objectives: performance, fairness, and predictability. We discuss scheduling algorithms for performance improvement in Section VI.1.1. We discuss algorithms for fairness and predictability in Section VI.1.2.

### VI.1.1 Scheduling for performance optimization

Snavely et al. [68] concurrently with Parekh et al. [59] introduced symbiotic job scheduling. Symbiotic job scheduling optimizes contention for shared resources on a multithreaded core. Symbiotic scheduling algorithms use online performance monitoring and heuristics to determine the thread schedules with the least resource contention. Snavely and Parekh evaluated their algorithms using user level prototypes. These algorithms optimized performance via co-scheduling threads to minimize contention for shared resources on a multithreaded core (i.e. functional units, L1 caches, etc.). Snavely's algorithm was shown to improve the overall system throughput by 40% and the turnaround time for a job mix by 33% [68]. Parekh's algorithms achieved the overall speedup of 7-15%.

Nakajima et al. [56] implemented a user-level scheduling utility for Linux that ensures that threads competing for the core's resources are not scheduled on the same core.

Siddha et al. described a chip-multithreading-aware extension to the Linux load-balancing algorithm [66]. The scheduler balances the assignment of threads among the cores and thread contexts, balancing shared resources on the core (i.e., functional units and L1 caches) and on the chip (i.e., the L2 cache, the memory bus).

Symbiotic job scheduling and load-balancing pursue performance optimizations, similar to our non-work-conserving and target-miss-rate algorithms. The difference is that we focus on optimizing contention for the L2 cache, and this has directed the design of our algorithms. Our non-work-conserving and target-miss-rate algorithms differ from the symbiotic job scheduling algorithm in that we use performance models to estimate contention for the L2 cache, while the symbiotic algorithms use heuristics.

Our non-work-conserving algorithm pursues an objective similar to the compiler optimization presented by Jung et al. [40]. Jung's optimization determines the optimal number of threads to execute a parallelizable loop on a multithreaded core. While Jung et al. built their model to consider all shared resources, they observe, similarly to us, that the L2 cache has a higher impact on performance than other resources. The key parameter for Jung's model is the workload's instruction mix, while our model relies on the perfect-cache IPC and the miss rate.

### VI.1.2 Scheduling for fairness and predictability

We now discuss work addressing fairness and predictability caused by unequal cache sharing on chip multithreaded processors. This work is relevant to our cache-fair algorithm. We discuss both hardware and software approaches, as it is not yet clear which approach or whether a combination of the two will yield the best solution.

Previous research investigated multicore processor architectures that enforce fair resource sharing or expose control over resource allocation to the operating system [22,29,42,60,73]. Some of these approaches use dynamic hardware resource partitioning; policies guiding partitioning decisions are either built into the hardware or controlled by the operating system via priority assignment. These architectural solutions pursued performance predictability [22] and fairness [42,60], as well as performance [29,73]. Hardware has direct control over resource allocation and could eliminate the need for complex and potentially inaccurate performance modeling in the software. The downside of a hardware solution is higher development cost and longer time-to-market. To the best of our knowledge, none of the proposed hardware architectures has been made commercially available. Our scheduling algorithm runs on systems that exist today, allowing applications to benefit from the performance and power of multicore processors while at the same time enjoying fairness and performance predictability of conventional processors.

Existing software solutions addressing performance predictability use co-scheduling [39]. Our algorithm uses a novel approach. It improves performance predictability by allocation of CPU cycles – a common mechanism used in CPU scheduling. Our approach is superior to co-scheduling, because with co-scheduling if the system cannot find the "right" co-runner for the thread, the thread's performance remains co-runner dependent. Additionally, co-scheduling requires estimating how a thread's performance is affected by a particular co-runner. Previous work on co-scheduling used performance models and heuristics. The effectiveness of these models and heuristics has been demonstrated on systems running at most four concurrent threads. It is an open question whether these methods scale for multicore systems with larger degrees of concurrency, such as 32 (already available today) [44]. Our cache-fair algorithm does not have inherent scalability limitations.

## VI.2 Analytical Performance Modeling

This section describes relevant work in performance and cache modeling. Section VI.2.1 describes multithreaded processor IPC models relevant to the non-work-conserving and cache-fair algorithms. Section VI.2.2 describes cache models relevant to all three algorithms.

### VI.2.1 Processor IPC models

We use models for a processor's instructions per cycle (IPC) in the non-work-conserving and the cache-fair scheduling algorithms.

The IPC model in the cache-fair scheduler estimates the IPC for a single-threaded core, and is similar to several existing models [53,69,76], with straightforward adaptations to suit our processor architecture.

In the non-work-conserving algorithm, we use the IPC model for multithreaded cores. Similar models were proposed by Saavedra-Barrera et al. [62] and Dubey et al. [30]. The main difference between the earlier models and ours is in the purpose and design. Our model is designed to work online inside a thread scheduler, so it is simple, uses basic probability theory, and has a trivial closed-form solution. Previous models are designed for offline studies of processor architectures, so they are more complex: they use Markov processes and have only approximate closed form solutions. Despite the simplicity of our model, its accuracy is comparable to these more complex models (for homogeneous multithreaded workloads). Another distinction of our model is that it works with inputs obtainable at runtime and furnished by a compiler, although some parameters are derived offline.

Offline parameter derivation could be a limitation of our scheduling algorithm in a practical setting, and it is worth investigating techniques for online derivation of these parameters. Recent work suggests avenues for such exploration. Moseley et al. presented several cross-architectural IPC models for multithreaded processors that work online and do not require offline derived parameters [55]. Their models have lower accuracy than ours: they model all sources of resource contention with simple models derived using linear regression and recursive partitioning. It is worth investigating whether we could use some of their techniques to eliminate the need for offline parameter derivation in our model.

Jung et al. proposed an IPC model for multithreaded processors used by a compiler to determine the optimal number of threads to execute a parallelizable loop [40]. This IPC model is parameterized by the instruction mix. We observed that our IPC model was less accurate for workloads whose instruction mix was dissimilar from the instruction mix of the programs used to train the model. Incorporating techniques used in Jung's model could help improve our model's accuracy.

### VI.2.2 Cache models

Our contribution to the domain of cache modeling is adapting the single-threaded reuse-distance model to work for multithreaded workloads on shared-cache processors. Chandra et al., working concurrently with us, proposed similar adaptations [23]. While we applied our multithreading adaptations to the Berg-Hagersten reuse-distance model [18], Chandra et al. applied theirs to the stack-distance model [21]. The stack-distance model is similar to the reuse-distance model (both models use a workload's memory reuse profile), so these adaptations pursue a similar goal. Two of the adaptations presented in Chandra's work, *frequency of access (FOA)* and *stack-distance competition (SDC)* are similar to our COMB and COMB+IPC methods. Chandra's FOA and SDC models produce similar accuracy to COMB and COMB+IPC. Chandra et al. also present a probability-based PROB model that has higher accuracy, but uses floating point operations and takes more steps to compute.

Suh et al. designed an accurate cache model for shared-cache systems that uses a workload's *miss-rate curve* to estimate its miss rate [72]. Miss-rate curves are difficult to obtain online. The authors presented a new hardware extension for generating miss-rate curves [73]. We opted not to rely on special hardware.

## VI.3 Summary

Our work addresses performance, predictability and fairness on chip multithreaded processors using novel approaches. Our algorithms do not rely on co-scheduling and thus have better potential for scalability. Our algorithms do not rely on special hardware and, therefore, work on existing systems. We contributed to the domain of analytical performance modeling by presenting several new models targeted specifically for online use, and evaluated trade-offs between accuracy and efficiency of these models. Few of the chip-multithreading-aware scheduling algorithms described in the literature have been implemented in real operating systems. We implemented two of our algorithms in a commercial operating system. This led us to develop unique insight on practical considerations for such algorithms, such as the need for efficient online models and phase-detection algorithms suitable for chip multithreaded processors.

# Chapter VII. SUMMARY

We conclude this dissertation by summarizing its lessons and contributions and discussing how the pieces of our work relate to each other. We follow with a discussion of the role of hardware in addressing the problems of performance, predictability and fairness on chip multithreaded systems. We then provide insight into designing software and hardware for future chip multithreaded systems. We end with a closing note.

## VII.1 Lessons, Contributions, and the Big Picture

This dissertation described three scheduling algorithms for chip multithreaded processors. These algorithms considered sharing a performance critical resource, the L2 cache, in pursuing three key objectives: *optimal performance, fairness, and predictability.* While these objectives could be viewed as orthogonal, we found that they are usually tightly coupled, and pursuing one objective requires a trade-off with another.

An orthogonal view on performance and fairness could be expressed by combining the non-work-conserving and the cache-fair policies. For example, a system could use the non-work-conserving algorithm to prevent thrashing, and then also use the cache-fair algorithm to ensure fairness and performance predictability.

However, performance and fairness are not always orthogonal, as we learned from our experience with the target-miss-rate algorithm. The target-miss-rate algorithm dramatically improves system throughput (by as much as 46%), but sacrifices fairness for high-miss-rate threads, decreasing their CPU share. Our cache-fair algorithm remedies this effect: it significantly improves fairness, while at the same time increasing the throughput. Balancing fairness and performance is a challenging problem, and we learned that while excessive focus on throughput can significantly hurt fairness, focusing on fairness actually improves throughput. Kim et al, who designed a hardware scheme for improved cache sharing, arrived at a similar conclusion [42]. An interesting avenue of future work is to identify whether this rule has applicability to a wider system domain.

One other lesson we learned from the target-miss-rate algorithm is that the operating system scheduler can effectively control performance on chip multithreaded systems via adjustments to threads' CPU shares. This led us to realize that we can use this control to improve performance predictability, inspiring the design of the cache-fair algorithm

The cache-fair algorithm reduces co-runner dependent performance variability from a dramatic 28% to a negligible 4% across the board. The algorithm is equipped with mechanisms that carefully balance performance and fairness: The algorithm may mildly affect the performance of best-effort threads, but it lets the user classify threads, ensuring that it does not penalize critical applications.

Since the advent of multiprogramming, fairness has become an important consideration in system design. Chip multithreading processors ignore fairness, and this has generated a number of research systems addressing fairness [22,39,42,60]. We believe that our cache-fair algorithm is a unique point in a spectrum of solutions. We do not rely on special hardware – and this makes our solution ready for use on existing systems. We do not rely on co-scheduling – and this makes our algorithm robust and scalable.

## VII.2 The Role of Hardware

The problems addressed in this dissertation are unique to chip multithreaded processors, so we must ask whether we should address these issues in hardware. While we cannot yet provide a definite answer, we discuss factors that one must consider when thinking about the role of hardware.

First of all, hardware solutions are, in general, more costly than software solutions. They have a longer time to market, and once put in place they are more difficult to change and tune. On the other hand, addressing a hardware deficiency in software may cause the software to become bloated and complex, introducing bugs and lowering maintainability. So the question that we must ask is *what is the right* *interface between the software and the hardware on chip multithreaded systems.*

Our view is that the hardware must implement mechanisms, not policies. It must be up to the software to decide which policy to apply, but the hardware must provide convenient mechanisms for enforcing the policy. Only in cases where a policy is so pervasive that ALL systems use it, should the policy be built into the hardware[8].

Unfortunately, chip multithreaded processors have not followed the "thou-shalt-not-enforce-policies" design principle. Modern chip multithreaded processors implement a throughput-maximizing policy: they allocate shared resources in an ad-hoc fashion, depending on the needs of co-runners [60]. As a result, a more aggressive co-runner will use more cache lines or will issue more instructions into the processor's pipeline. This has caused problems with both performance and fairness.

The hardware should provide simple mechanisms on which the software implements its policies. The key challenge is to make these mechanisms simple enough so that they are actually useful.

The algorithms described in this dissertation were implemented entirely in software. We believe, however, that the implementations could be made simpler if better hardware support were available. For example, hardware counters summarizing cache-reuse patterns [73] could simplify

[8] An example of a pervasive system policy migrating into the hardware is TCP offloading.

miss-rate modeling. However, it would be unwise to implement our algorithms entirely in hardware. Not only would this result in prohibitive complexity, this would also limit the flexibility of tuning the algorithms. Therefore, we believe that even as hardware evolves to provide better support for resource management on chip multithreaded processors, the role of software solutions will continue being important.

## VII.3 A Glance into the Future

The emergence of chip multithreaded architectures forces us to reconsider system design. This dissertation showed adaptations to scheduling algorithms and analytical performance models and argued for the need to adapt online phase-detection algorithms. The advent of chip multithreaded hardware will spur other major system transformations.

Systems built on top of conventional processors have established layers of abstractions and interfaces used by applications to interact with the system's software and hardware. Systems built on chip multithreading platforms may require reconsideration of abstraction layers, producing new cross-layer system designs. Chip multithreaded architectures are sufficiently different from conventional processors that we cannot simply use the old ways of building operating systems and applications on top of them. Neither can we maintain the same old interfaces between the hardware and the operating system, and between the operating system and applications.

We designed our scheduling algorithms by making the operating system bridge the gap between the applications and the hardware, while allowing the applications and the hardware to remain unchanged. Even though this design has benefits, we believe that our solutions could be made simpler if the operating system could interact more closely with the applications and the hardware.

For example, we discovered that current techniques for modeling cache miss rates on chip multithreaded processors have either limited accuracy or high performance overhead. Hardware counters summarizing application cache-access patterns [73] and allowing to obtain stack-distance or reuse-distance profiles online could enable accurate and efficient modeling of cache miss rates. Alternatively, the application itself, with a help of a special library or a runtime system, such as a Java Virtual Machine, could provide the information to assist miss rate modeling.

The hardware should tell the operating system how its resources, such as the cache and the memory bus, are shared among the co-runners. Intel's Core Duo has hardware core-specific counters for L2 cache allocation [37], but other processor models do not furnish such information.

Applications possess a great deal of information that could aid the system in its decisions. On chip multithreaded platforms, performance depends on how application threads interact with each other. While in some cases the operating system or the hardware can detect these interactions [59,68], there are scenarios where the only way to learn about them is directly from the application. For example, in transactional memory systems [26], threads that conflict on transactions could experience better performance if the operating does not co-schedule them. Another example is applications where threads share code and data. Placing such threads in close physical proximity to each other (i.e., on the same core of a multi-core processor, or on the same chip of the chip multithreaded multiprocessor) will produce lower communication latency and better performance. While code and data sharing could be detected by observing coherency traffic on cache and memory interconnects, getting hints directly from the application could be a simpler solution.

## VII.4 A Closing Note

The advent of chip multithreaded processors has encouraged reconsideration of operating system design. On chip multithreaded processors, resource sharing and, as a result, application performance depends on the interactions among the co-scheduled threads. The operating system is the entity controlling scheduling, thus having an important effect on system performance. Therefore, the operating system will continue to play a major role in successful adoption of chip multithreaded processors.

As chip multithreaded hardware evolves to expose control over resource sharing to the operating system, software solutions will become simpler and more effective. Nevertheless, the role of software will not diminish. Years of practice have demonstrated the benefit of controlling resource sharing in the operating system and keeping the hardware simple [74]. This dissertation has furthered the knowledge of operating system design for chip multithreaded processors and paved avenues for future research.

# Chapter VIII. REFERENCES

[1] AMD: Multi-core Processors - the Next Evolution in Computing. *http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf*, 2006

[2] IBM Virtualization. *www.ibm.com/virtualization*, 2006

[3] Intel Core Duo Processors. *http://www.intel.com/products/processor/coreduo/*, 2006

[4] mpiBLAST: Open-Source Parallel Blast. *http://mpiblast.lanl.gov*, 2006

[5] POWER4 System Microarchitecture. *http://www-03.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html*, 2006

[6] SPEC CPU2000 web site. *http://www.spec.org*, 2006

[7] Sun Grid Frequently Asked Questions (#14). *http://www.sun.com/service/sungrid/faq.xml#q14*, 2006

[8] The listing of performance counters available on SPARC T1. *http://cvs.opensolaris.org/source/xref/on/usr/src/uts/sun4v/cpu/niagara_perfctr.c*, *http://cvs.opensolaris.org/source/xref/on/usr/src/uts/sun4v/pcbe/niagara_pcbe.c*, 2006

[9] Advanced Micro Devices. AMD Demonstrates Dual Core Leadership. *http://www.amd.com*, 2004

[10] A. Agarwal, J. Hennessey, and M. Horowitz. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184-215, 1989

[11] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium On Computer Architecture (ISCA),* pp. 104-114, June 1990

[12] Anastassia Ailamaki, David J DeWitt, Mark D. Hill, and David A. Wood. DBMSs on modern processors: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases,* pp. 266-277, September 1999

[13] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-Time Scheduling on Multicore Platforms. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium,* pp. 179-190, April 2006

[14] Gary Anthes. Hard Cores: Multicore chips provide power but make app development tough. *http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=112303,* 2006

[15] D. A Bader, V. Sachdeva, V. Agarwal, G. Goel, and A. N. Singh. BioSPLASH: A Sample Workload for Bioinformatics and Computational Biology for Optimizing Next-Generation Performance Computer Systems. *Technical Report, University of New Mexico,* 2005

[16] Rajeev Balasubramonian, Sandhya Dwarkadas, and David Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the 30th International Symposium on Computer Architecture,* pp. 275-287, 2003

[17] Luiz A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Quadeer, B. Sano, S. Smith, A. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA),* pp. 282-293, 2000

[18] Erik Berg and Erik Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 1-8, 2004

[19] E. Berger, S. Kaplan, B. Urgaonkar, P. Sharma, A. Chandra, and P. Shenoy. Scheduler-Aware Virtual Memory Management. *Poster, Symposium on Operating Systems Principles (SOSP)*, 2003

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720-748, 1999

[21] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Proceedings of the 12th Intl.Workshop on Languages and Compilers for Parallel Computing*, pp. 365-379, 1999

[22] F. J. Cazorla, Peter M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors. In *Proceedings of the 1st Conference on Computing Frontiers*, pp. 433-443, 2004

[23] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Multi-Processor Architecture. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pp. 340-351, 2005

[24] Y. Chen, Q. Diao, C. Dulong, C. Lai, W. Hu, E. Li, W. Li, T. Wang, and Y. Zhang. Performance Scalability of Data-Mining Workloads in Bioinformatics. *Intel Technology Journal*, 9(2), 2005

[25] Richard Cole, Alan M. Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andrea W. Richa, Ramesh K. Sitaraman, and Eli Upfal. On Balls and Bins with Deletions. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, pp. 145-158, 1998

[26] Peter Damron, Alexandra Fedorova, Yosef Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems,* October 2006

[27] P. J. Denning. Thrashing: Its Causes and Prevention. In *Proceedings of the AFIPS Fall Joint Computer Conference,* pp. 915-922, 1968

[28] Ashutosh S. Dhodapkar and James E. Smith. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture,* pp. 217-227, 2003

[29] G. Dorai and D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT),* pp. 30-41, 2002

[30] P. K Dubey, A. Krishna, and Mark S. Squillante. Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures. In *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems,* pp. 110-122, 1995

[31] Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23d Annual International Symposium On Computer Architecture (ISCA),* pp. 203-212, May 1996

[32] W. Feller. An Introduction to Probability Theory and Its Applications. *John Wiley and Sons*, 1968

[33] M Funk. Simultaneous Multi-threading (SMT) on eServer iSeries Power5 Processors. *http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/pdf/SMT.pdf,* 2004

[34] Mark Hachman. Intel Cancels 4-GHz Pentium 4. *E-week Channel Insider, http://www.thechannelinsider.com,* October 2004

[35] Sebastien Hily and Andre Seznec. Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading. In *Proceedings of the Workshop On Multi-Threaded Execution, Architecture and Compilation,* January 1998

[36] Intel Corporation. *http://www.intel.com/research/silicon/mooreslaw.htm,* 2005

[37] Intel Corporation. Pentium 4 Programmer's Manual. *http://sunsite.rediris.es/pub/mirror/intel/Pentium4/manuals/25366919.pdf,* 2006

[38] Intel Corporation. Superior Performance with Dual-Core. *ftp://download.intel.com/products/processor/xeon/srvrplatformbrief.pdf,* 2006

[39] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS),* pp. 134-145, 2002

[40] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 236-246, 2005

[41] Ron Kalla, Balaram Simharoy, and Joel M. Tendler. IBM POWER5 Chip: A Dual-core Multithreaded Processor. *IEEE Micro*, 24(2):40-47, March 2004

[42] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT),* pp. 111-122, 2004

[43] L. Kleinrock. Queuing Systems . *Wiley*, 1975

[44] Poonacha Kongetira. A 32-way Multithreaded SPARC(R) Processor. In *Proceedings of the 16th Symposium On High Performance Chips (HOTCHIPS),* August 2004

[45] James Laudon. Performance/Watt: the New Server Focus. *ACM SIGARCH Computer Architecture News*, 33(4):5-13, November 2005

[46] James Laudon, A. Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference On Architectural Support For Programming Languages And Operating Systems (ASPLOS),* pp. 308-318, October 1994

[47] Benjamin Lee. An Architectural Assessment of SPEC CPU Benchmark Relevance. *Harvard University Technical Report TR-02-06,* 2006

[48] Jack L. Lo, Luiz A. Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium On Computer Architecture (ISCA),* pp. 39-50, July 1998

[49] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca M. Stamm, and Dean M. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transations on Computer Systems*, 15(3):322-354, August 1997

[50] Peter S. Magnusson, Fredrik Dahlgren, Hakan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrom, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of the USENIX Annual Technical Conference,* June 1998

[51] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Allan Miller, and Michael Upton. Hyper-threading

Technology Architecture and Microarchitecture. *Intel Technical Journal*, 6(1):4-15, February 2002

[52] Takumi Maruyama. SPARC64 VI: Fujitsu's Next Generation Processor. In *Proceedings of the Microprocessor Forum*, October 2003

[53] R. Matick, T. J Heller, and M. Ignatowski. Analytical Analysis of Finite Cache Penalty and Cycles per Instruction of a Multiprocessor Memory Hierarchy Using Miss Rates and Queuing Theory. *IBM Journal Of Research And Development*, 45(6):819-842, 2001

[54] Richard McDougall and Jim Mauro. Solaris Internals:Core Kernel Architecture. *Sun Microsystems Press*, 2001

[55] Tipp Moseley, Joshua L. Kihm, Daniel A. Connors, and Dirk Grunwald. Methods for Modeling Resource Contention on Simultaneous Multithreading Processors. In *Proceedings of the International Conference on Computer Design*, pp. 373-380, 2005

[56] Jun Nakajima and Venkatesh Pallipadi. Enhancements for Hyper-Threading Technology in the Operating System - Seeking the Optimal Scheduling. In *Proceedings of the Second Workshop on Industrial Experiences with Systems and Software*, December 2002

[57] Daniel Nussbaum, Alexandra Fedorova, and Christopher Small. The Sam CMT Simulator Kit. *Sun Microsystems TR 2004-133,* March 2004

[58] R. Onvural. Survey of Closed Queuing Networks With Blocking. *ACM Computing Surveys*, 22(2):83-121, 1990

[59] Sujay Parekh, Susan J. Eggers, and Henry M. Levy. Thread-Sensitive Scheduling for SMT Processors. *University of Washington Technical Report 2000-04-02,* April 2004

[60] S. E. Raasch and S. K. Reinhardt. Applications of Thread Prioritization in SMT Processors. In *Proceedings of the Workshop On Multi-Threaded Execution, Architecture and Compilation,* 1999

[61] E. Rosti, E. Smirni, G. Serrazi, and L. Dowdy. Analysis of Non-Work-Conserving Processor Partitioning Policies. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing,* pp. 165-181, 1995

[62] R. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 169-178, 1990

[63] Alex Settle, Joshua L. Kihm, Andrew Janiszewski, and Daniel A. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT),* pp. 63-73, 2004

[64] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 165-176, 2004

[65] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 45-57, 2002

[66] Suresh Siddha and Venkatesh Pallipadi. Chip Multi Processing Aware Linux Kernel Scheduler. In *Proceedings of the Linux Symposium,* v.2 pp.193-203, 2005

[67] E. Smirni, E. Rosti, G. Serrazi, L. Dowdy, and K. C Sevcik. Performance Gains From Leaving Idle Processors in Multiprocessor Systems. In *Proceedings of the International Conference on Parallel Processing,* pp. 203-210, 1995

[68] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 234-244, November 2000

[69] Yan Solihin, V. Lam, and Josep Torrellas. Scal-Tool:Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing,* pp. 17-30, 2006

[70] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic Evaluation of Shared-memory Systems with ILP Processors. In *Proceedings of the 25th Annual International Symposium On Computer Architecture (ISCA),* pp. 380-391, 1998

[71] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture,* pp. 248-252, February 2005

[72] G. E. Suh, S Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing,* pp. 1-12, 2001

[73] G. E. Suh, S Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture,* pp. 117-128, 2002

[74] Andrew S. Tanenbaum. Modern Operating Systems. *Prentice Hall,* 2001

[75] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium On Computer Architecture (ISCA),* pp. 392-403, June 1995

[76] Harvey J. Wasserman, Olaf M. Lubeck, Yong Luo, and Frederico Bassetti. Performance Evaluation of the SGI Origin2000: A Memory-Centric Evaluation of LANL ASCI Applications. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing,* pp. 1-11, 1997

[77] D. Willick and D. Eager. An Analytical Model of Multistage Interconnection Networks. In *Proceedings of the ACM SIGMETRICS,* pp. 192-199, 1990

[78] W. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications Of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20-24, March 1995

# Chapter IX. APPENDIX

## IX.1 A Model for Memory-bus Delay

Our goal is to model *memory-bus delay* – the time that a memory request waits for the memory bus to become available. We begin the presentation of our model by describing how we represent the memory system. This will bring us to the key concept in our model, a *request cycle window.*

We represent the memory system as a server that answers requests for values stored in main memory. In our simulated system, there are five concurrent streams of memory requests originating from four virtual processors and one write buffer. We refer to these request originators as *consumers*, and the number of request originators as *NUM_CONSUMERS.*

After issuing a memory request the consumer waits until the memory bus becomes available, for the amount of time corresponding to the memory-bus delay. Once the bus is available, the consumer reserves it for a period of *WIRE_TIME* cycles. *WIRE_TIME* is fixed; we compute it by dividing the size of the memory request, which equals the size of the L2 cache line, by the memory-bus bandwidth. Once the memory request is serviced, the consumer spends some time performing computation before making another request. We call the combination of *WIRE_TIME* and compute time the *request cycle window* (*REQUEST_CYCLE_WINDOW*) – this is the number of cycles between the beginning of service for a request (after the bus has been reserved) and the arrival of the next request. Figure A-1 illustrates this.

### IX.1.1 Request Cycle Window

The size of *REQUEST_CYCLE_WINDOW* determines the degree of competition for the memory bus. *REQUEST_CYCLE_WINDOW expires* upon the arrival of a new memory request. A shorter window implies a higher request arrival rate and a more intense memory-bus competition. Since *WIRE_TIME* is fixed, the size of *REQUEST_CYCLE_WINDOW* is determined by the compute time – the time that the consumer spends between consecutive memory requests. We now show how we compute the length of *REQUEST_CYCLE_WINDOW* and then explain how we use it to model the memory-bus delay.

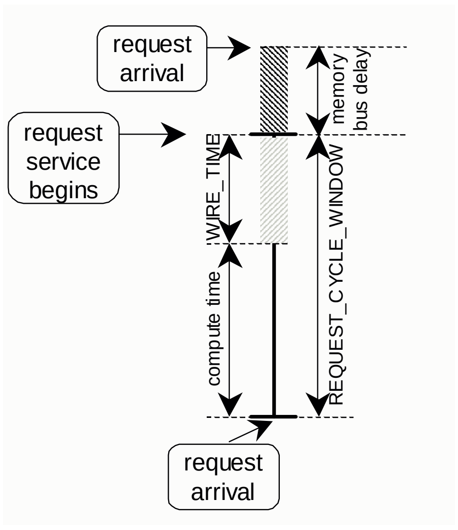


**Figure A-1. Request cycle window**

*REQUEST_CYCLE_WINDOW* is equivalent to the number of cycles that elapse between the arrivals of two consecutive memory requests from the same consumer when there are no memory-bus delays (See Figure A-1). We can trivially compute this quantity using a consumer's L2 cache misses per cycle under unlimited memory bandwidth $L2\_MPC_c$ (because L2 cache miss rate corresponds to memory request rate). The inverse of $L2\_MPC_c$, i.e., the number of cycles between two consecutive cache misses (or memory requests), equals to *REQUEST_CYCLE_WINDOW*.

We compute a consumer's $L2\_MPC_c$ as follows:

$$L2\_MPC_c = \frac{(L2\_COMB\_RMR + L2\_COMB\_WMR) * IPC}{NUM\_CONSUMERS} \qquad \text{(A1)},$$

where *L2_COMB_RMR* and *L2_COMB_WMR* are the L2 read miss rate and L2 write miss rate including write-back rates[9], and *IPC* is the aggregate IPC for the multithreaded workload estimated using the unlimited-bandwidth IPC model[10]. We divide the quantity in the numerator by the number of consumers in order to compute the $L2\_MPC_c$ for a single consumer.

*REQUEST_CYCLE_WINDOW* is the inverse of $L2\_MPC_c$.

[9] These quantities are defined in Chapter III.
[10] See Equation III.7.

$$REQUEST\_CYCLE\_WINDOW = \frac{1}{L2\_MPC_c} \qquad \text{(A2).}$$

## IX.1.2 Computing the memory-bus delay

Now, let us show how to compute the memory bus delay using *REQUEST_CYCLE_WINDOW*. We model the memory-bus delay from the point of view of an individual consumer, thread *t0*, and we assume that all consumers experience identical delays. This assumption holds for the consumers representing virtual processors, because we implemented our non-work-conserving algorithm for the environment where all threads do the same work. The assumption does not hold for the consumer representing the write buffer, since the write buffer may issue requests at a different rate than the virtual processors, but making this assumption considerably simplifies the model.

We reason that if all other consumers send their requests at the same non-bursty rate as *t0*, those consumers' requests will arrive to the memory system sometime during *t0*'s *REQUEST_CYCLE_WINDOW*. As a result, the amount of time that *t0* has to wait for the memory bus once its *REQUEST_CYCLE_WINDOW* expires depends on a) the size of its *REQUEST_CYCLE_WINDOW* and b) at which position in this window the other consumers' requests arrive. Intuitively, if the other consumers' requests arrive early in *t0*'s window, they may have time to finish before *t0*'s window expires. In this case, *t0* will not have to wait for the memory bus when it sends its next request. On the other hand, if the other requests arrive late, then they will not finish before *t0*'s window expires, and *t0* will have to wait for the bus when it sends its next request.

We divide *REQUEST_CYCLE_WINDOW* into two portions: the *top portion* and the *bottom portion*, where the top portion corresponds to the early arrival of the other consumers' requests, and the bottom portion corresponds to late arrivals.

We model the memory-bus delay by estimating the wait times for the top and bottom portions, computing the probabilities that the requests' arrival falls into a particular portion, and then weighing the delay in each portion by that portion's probability. Figure A-2 illustrates how we divide the window into the portions and how we estimate the delays for each portion. We now describe this in detail.

The top portion starts at the top of the window. If the window is large enough such that the other consumers' requests can be serviced before *REQUEST_CYCLE_WINDOW* expires, the top portion stretches until the latest point at which those requests must arrive so that they free the wire before the window expires. Figure A-2a illustrates this case. If the other consumers' requests arrive in the interval of time covered by the top portion of the window, *t0*'s memory-bus wait time will be zero.

However, if the window is not large enough, then the top portion covers the stretch of the window when the wire is still reserved by *t0*'s memory request – this is the top interval of the window equal to *WIRE_TIME* (Figure A-2b). In this case, *t0* will need to wait for the duration of time equal to:

$$WIRE\_TIME * NUM\_CONSUMERS - REQUEST\_CYCLE\_WINDOW \,,$$

once its next request arrives.

We compute the wait times associated with the top portion of the window using the following formula:

$$WAIT\_TOP = MAX(0, WIRE\_TIME * NUM\_CONSUMERS - REQUEST\_CYCLE\_WINDOW) \qquad \text{(A3).}$$

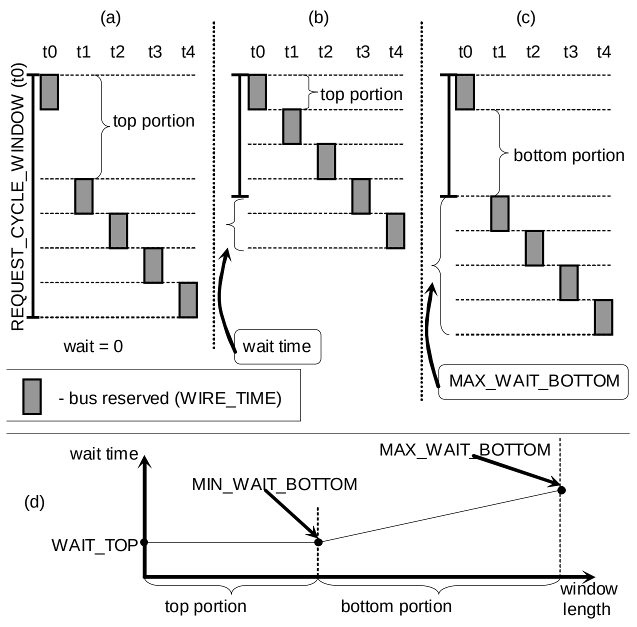


**Figure A-2. Illustration of different wait times.**
Memory-bus wait time for *t0* depends on how the requests of the other four consumers are positioned in its *REQUEST_CYCLE_WINDOW*. (a) *t0*'s wait time is zero: *t0*'s window is large enough, so that other consumers' requests are served before *t0*'s window expires; (b) shows the minimum time that *t0* has to wait if its window is not large enough; (c) shows the maximum time *t0* has to wait if its window is not large enough.

The bottom portion of the window is the part of the window not covered by the top portion. The minimum amount of wait for the bottom portion equals to *WAIT_TOP* (Equation A3), because the bottom portion commences where the top portion ends. To understand what the maximum wait would be, consider Figure A-2c. The maximum wait for the bottom portion occurs if the other consumers' requests arrive just before the *REQUEST_CYCLE_WINDOW* expires. The wait in this case is equal to the amount of time it takes to service those consumers' requests, *WIRE_TIME * (NUM_CONSUMERS – 1)*. We summarize the minimum and the maximum wait times associated with the bottom portion below:

*MIN_WAIT_BOTTOM = WAIT_TOP*

*MAX_WAIT_BOTTOM = WIRE_TIME * (NUM_THREADS – 1).*

One difference between the top and the bottom portions is that the wait time in the top portion is fixed (the exact wait time depends in the window size). The wait time in the bottom portion ranges between *MIN_WAIT_BOTTOM* and *MAX_WAIT_BOTTOM* (Figure A2d). We compute the wait time corresponding to the bottom portion as the average of the minimum and maximum wait times:

$$WAIT\_BOTTOM = \frac{MIN\_WAIT\_BOTTOM + MAX\_WAIT\_BOTTOM}{2} \quad \text{(A4)}$$

Using these definitions of the top and bottom portions, the lengths of these portions can be computed as follows:

$$top\_portion = MAX(WIRE\_TIME; \\ REQUEST\_CYCLE\_WINDOW - NUM\_CONSUMERS * WIRE\_TIME)$$

$$bottom\_portion = REQUEST\_CYCLE\_WINDOW - top\_portion$$

And the probability of the requests arriving in a given portion is simply the fraction of the *REQUEST_CYCLE_WINDOW* that the portion occupies:

$$P(arrive\_at\_top) = \frac{top\_portion}{REQUEST\_CYCLE\_WINDOW}$$

$$P(arrive\_at\_bottom) = \frac{bottom\_portion}{REQUEST\_CYCLE\_WINDOW}$$

To compute the overall memory-bus delay per transaction (*MEM_BUS_DELAY*), we weight the wait times associated with the top and bottom portions by their respective probabilities:

$$MEM\_BUS\_DELAY = P(arrive\_at\_top) * WAIT\_TOP + \\ P(arrive\_at\_bottom) * WAIT\_BOTTOM \qquad \text{(A5).}$$

The quantity in Equation A5 expresses the expected memory-bus delay per L2 miss transaction. Using Equation A5 we were able to estimate memory-bus delays to within 12% of the actual delays, on average. The median error was 10%, and the largest error was 25%.

### IX.1.3 Factoring the memory-bus delay in the IPC model

We account for the memory-bus delay in the IPC model used in the non-work-conserving scheduler by augmenting the cost of the L2 cache miss (*L2_MCOST*) with the memory-bus delay. (Recall that *L2_MCOST* is the sum of memory-access time and the memory-bus delay.)

### IX.1.4 An alternative view on REQUEST_CYCLE_WINDOW

We defined *REQUEST_CYCLE_WINDOW* as the time elapsing between the beginning of a service for a request and the arrival of the next request. An alternative way to define *REQUEST_CYCLE_WINDOW* is the time between the arrivals of two consecutive requests[11]. Under this definition, the request cycle window *includes* the time a consumer waits for the memory bus. Therefore, we cannot compute *REQUEST_CYCLE_WINDOW* using the unlimited-bandwidth model, because that would not reflect the inclusion of the memory-bus delay in the length of the window.

Using this alternative definition for *REQUEST_CYCLE_WINDOW* we estimate the associated memory-bus delay via the following iterative process:

0) Compute the size of *REQUEST_CYCLE_WINDOW* using the estimated unlimited-bandwidth IPC. Estimate the corresponding

[11] We used this alternative definition in our prototype implementation of the non-work-conserving scheduler.

memory-bus delay using the model presented in this section. Assign the estimated memory-bus delay to *previous_membus_delay.*

1) Augment *L2_MCOST* with *previous_membus_delay*. Estimate the IPC using the new *L2_MCOST*. Re-compute the size of *REQUEST_CYCLE_WINDOW* using the newly estimated IPC, and re-estimate the memory-bus delay *(new_membus_delay)*.
2) If *previous_membus_delay* roughly equals *new_membus_delay*, terminate the process and return *new_membus_delay.* Otherwise, set *previous_membus_delay* equal to *new_membus_delay,* repeat Step 1.

## IX.2 Modeling the Number of Write-Back Transactions

We present a model for estimating the number of read-triggered write-back transactions. Modeling this quantity is useful in situations when the processor does not provide a counter to measure this quantity directly. We assume that the processor has per-thread counters for the following statistics:

- the number of data read misses *(data_read_misses)*
- the number of instruction fetch misses *(ifetch_misses)*
- the number of write misses *(write_misses)*
- the total number of write-back transactions (*write_back_txn)*

Modern chip multithreaded processors are typically equipped with these counters [5,8,37]. In cases where the processor does not have the counter for *write_back_txn* we model this quantity using a counter for the number of cache line upgraded to exclusive ownership *(write_upgrades).*

The model consists of two steps: (1) estimating the total number of write-back transactions (if we cannot measure it directly) and (2) estimating the number of read-triggered write-back transactions.

<u>Estimating the total number of write-back transactions:</u> We compute the total number of write-back transactions *write_back_txn* generated by a thread as follows:

$$\mathit{write\_back\_txn} = \mathit{write\_misses} + \mathit{write\_upgrades}$$

When a thread allocates a cache line due to a write miss or acquires the exclusive ownership on a cache line it already owns, that thread intends to write that line for the first time. All dirty cache lines are eventually written to memory via write-back transactions, so we use the number of cache lines written for the first time to approximate the total number of write-back transactions.

<u>Estimating the number of read-triggered write-backs:</u> We now estimate how many of the total write-back transactions are read-triggered, i.e., triggered during data read misses and instruction fetch misses. We refer to the sum of data read misses and instruction fetch misses as *read_misses*:

$$read\_misses = data\_read\_misses + ifetch\_misses$$

We found that the fraction of read-triggered write-backs *(fraction_read_triggered)* is closely approximated by the fraction that read misses constitute of the total misses:

$$fraction\_read\_triggered = \frac{read\_misses}{(read\_misses + write\_misses)}$$

We estimate the number of read-triggered write-back transactions by multiplying the total number of write-back transactions by the fraction of read-triggered transactions:

**Table A-1. The number of write-back transactions (actual vs. estimated)**

| Benchmark | ACTUAL | ESTIMATED | DIFFERENCE (as % of actual) |
|---|---:|---:|---:|
| *art* | 2,818,878 | 2,849,604 | 1.09% |
| *crafty* | 53,060 | 50,882 | -4.11% |
| *gcc* | 164,406 | 158,737 | -3.45% |
| *gzip* | 98,846 | 101,162 | 2.34% |
| *mcf* | 1,986,911 | 2,631,937 | **32.46%** |
| *parser* | 252,466 | 295,715 | **17.13%** |
| *twolf* | 539,656 | 538,974 | -0.13% |
| *vortex* | 249,823 | 267,840 | 7.21% |
| *vpr* | 63,954 | 59,567 | -6.86% |

$$read\_triggered\_write\_backs = write\_back\_txn * fraction\_read\_triggered \quad \text{(A6).}$$

We validate our model by comparing the number of estimated read-triggered write-backs with the actual read-triggered write-backs for nine SPEC CPU2000 benchmarks. We measure the number of actual read-triggered write-backs by running each benchmark for 100 million instructions on our simulator configured with a single core and a 128KB L2 cache. We estimate the number of read-triggered write-backs by plugging the measured values for *data_read_misses, ifetch_misses, write_misses* and *write_upgrades* into Equation A6. Table A-1 compares the actual and estimated values.

Our model produces accurate estimates (within 7% of the actual values) for seven out of nine benchmarks. Our estimates are less accurate for *mcf* (32% of the actual) and *parser* (17% of the actual). For these benchmarks, the assumption that the total number of write-back transactions can be approximated by the sum of write misses and write upgrades (step 1 of our model) is not accurate. If we measure the total number of write-backs directly and then use step 2 to estimate the fraction of read-triggered write-backs, the accuracy for these two benchmarks improves. The estimated values are within 15% of the actual values for *mcf*, and within 5% for *parser.*

Intel's Core Duo processors provide counters for the number of write-back transactions [37], and given the popularity of Intel's processors it is reasonable to assume availability of such counters on many future chip multithreaded processors.

## IX.3 Phased Program Behavior

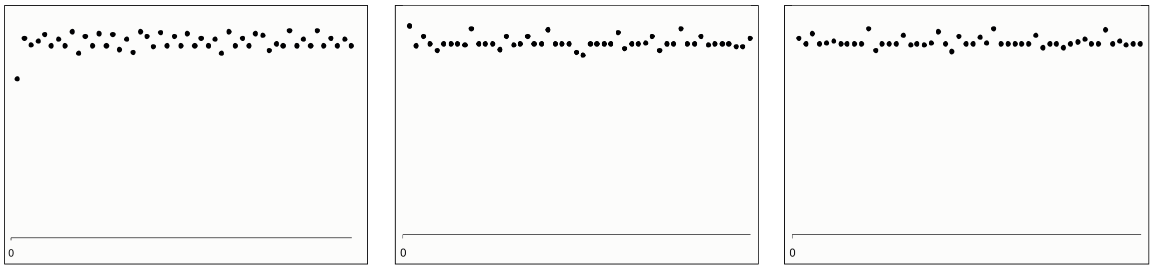
Programs typically exhibit phasing behavior throughout the execution [16,28,64,65] as their architectural profiles and runtime metrics change over the runtime. Our scheduling algorithms base their decisions on threads' behavior observed in the past, and, therefore, phase changes may affect the correctness of our algorithms. In this dissertation, however, we made the assumption that programs' L2 cache miss rates remain largely unchanged as the programs go through execution phases. While we believe that we could improve our algorithms by making them account for phase changes, in this section we demonstrate that for most of our benchmarks the L2 cache miss rate does not exhibit phasing behavior, supporting the validity of the experiments presented in the earlier chapters.

We analyze L2 cache miss rate behavior over *coarse-grained* time intervals, as opposed to *fine-grained* intervals. Fine-grained intervals are used to detect phases visible at a granularity of tens of thousands of instructions (i.e., a program's runtime metrics change during the n*th* 10K instructions compared to (n-1)*th* 10K instructions). Coarse-grained intervals are used to detect phases visible at a granularity of tens of *millions* of instructions. While existing phase detection research focused on fine-grained phases (e.g., Balasubramonian et al. detected phases lasting 10K-320K instructions [16]), due to the structure of our algorithms we focus on coarse-grained phases. Our algorithms measure and optimize performance over periods of tens of millions of instructions, and any finer-grained phase changes do not affect their correctness. For example, the cache-fair algorithm estimates a thread's fair miss rate over a period of 100 million instructions and then adjusts that thread's timeslice every 50 million instructions. While it is critical to detect phase changes observable at the granularity of 50 million instructions in order to avoid using an outdated estimate of the fair miss rate, it is not necessary to detect finer-grained phase changes.

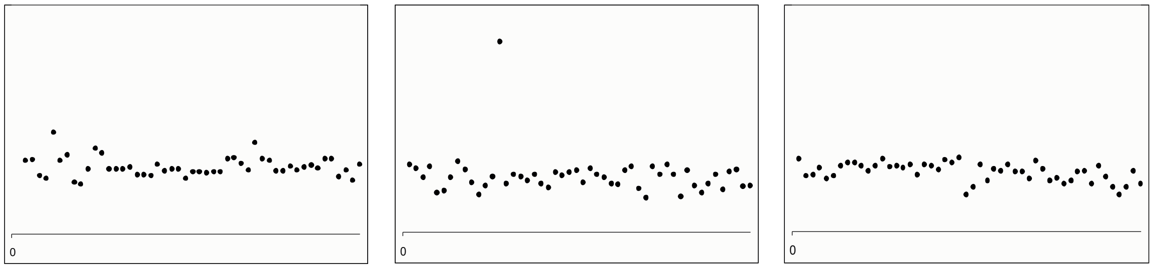
Figure A-3 demonstrates how L2 cache miss rate changes over time for the nine SPEC CPU2000 benchmarks used for the experiments with the cache-fair scheduler. For each benchmark there are three graphs. Each graph displays the number of L2 cache misses per instruction over time. Each dot represents the miss rate measured over a period of ten million instructions. Each graph shows the miss rates over a window of a half a billion instructions, so there are fifty observations on each graph. The first graph displays the half-billion instruction window immediately after the program starts, the second – after the program has run for one billion instructions, and the third – after the program has run for two billion instructions. Note that we deliberately do not label the axes to avoid polluting the graphs. We use the same scale (for x-axis and y-axis) for all graphs. What we emphasize here is how the miss rate changes over time, not the absolute values for the miss rates.

For all programs except *gcc* the L2 cache miss rate remains largely unchanged over time after some instability upon program initialization. Note the first graphs for *vortex* and *vpr*: the miss rate varies for at most 300 million instructions before stabilizing.

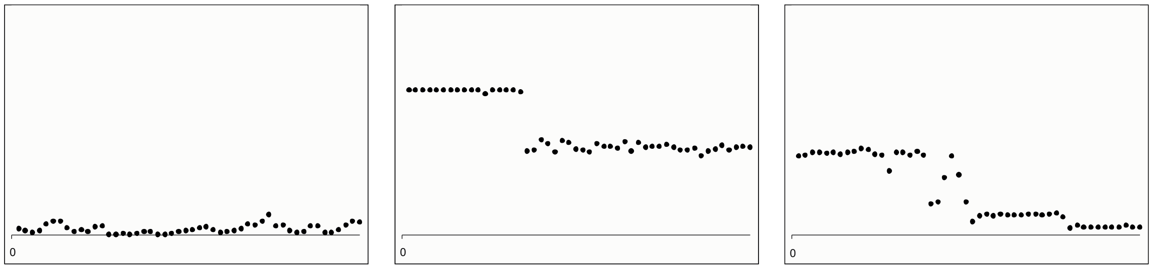
*Gcc*'s miss rate demonstrates phasing behavior: note the very different patterns of dots on all three graphs. *Gcc*, a compiler, performs different kinds of operations on the files over time, and this explains its changing miss rate. Some believe that servers and databases, whose behavior is dependent on user input, exhibit similar phased behavior [64]. While the assumption of stable miss rates is safe for many programs, algorithms for dynamic detection of program phase changes would improve robustness of algorithms such as ours.



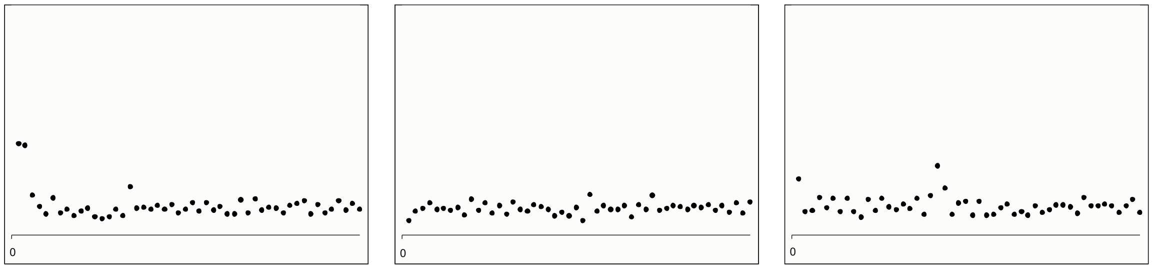
ART



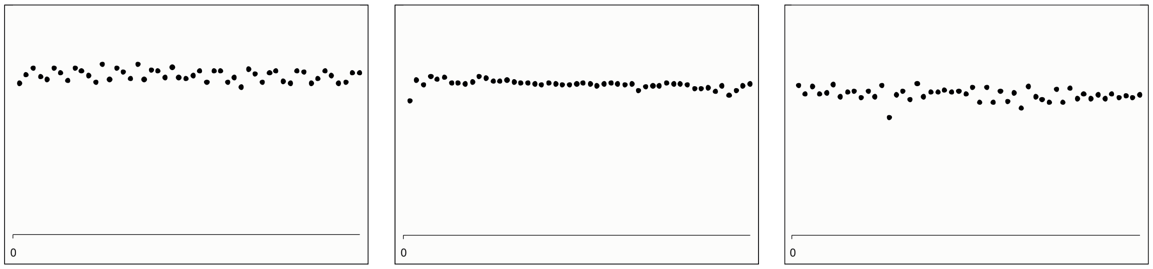
CRAFTY



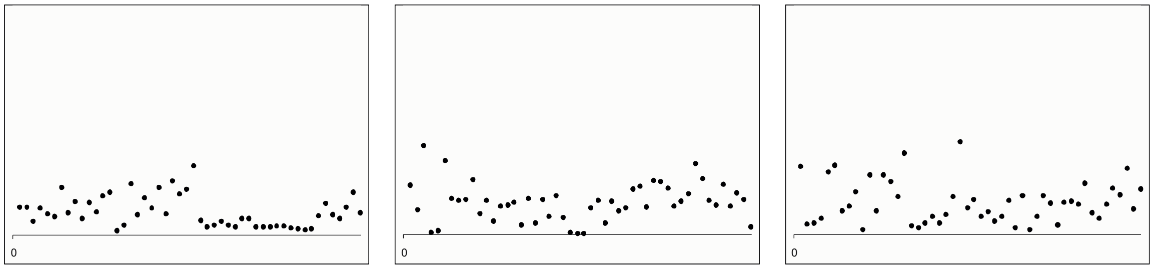
GCC



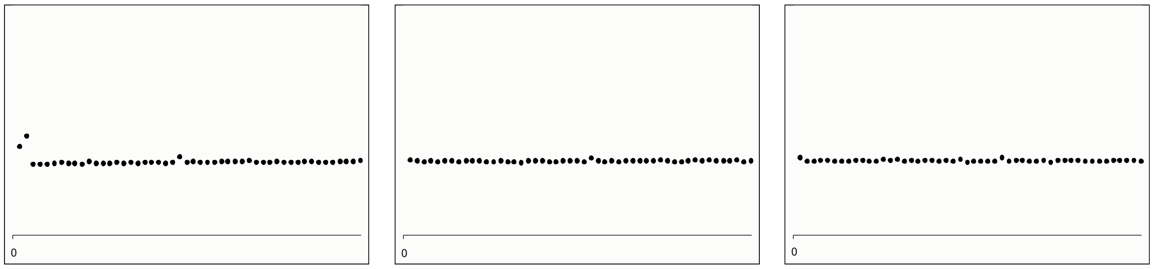
GZIP



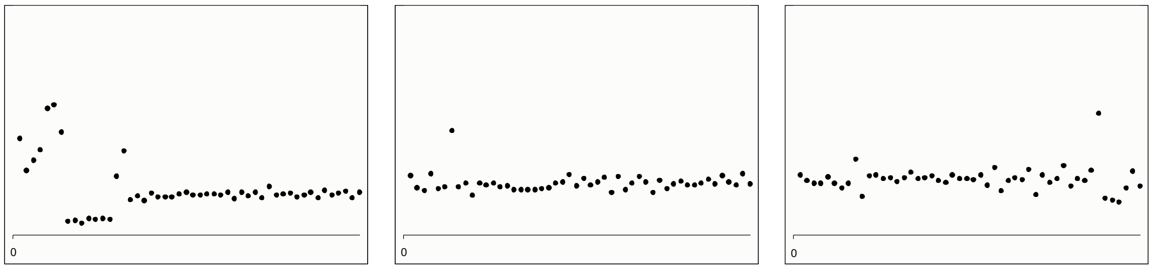
MCF



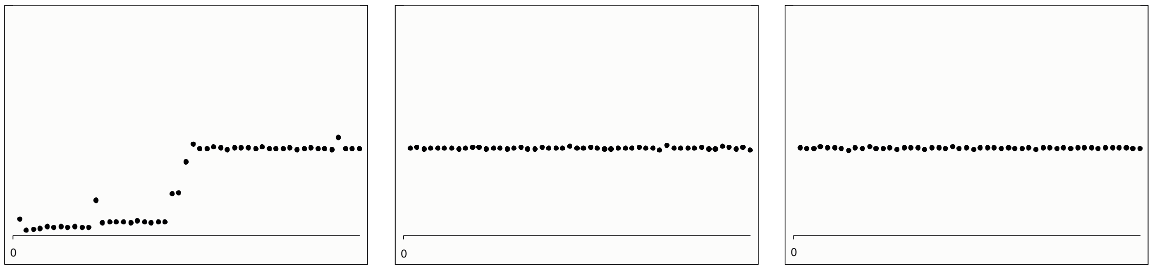
PARSER



TWOLF



VORTEX



VPR

**Figure A-3. L2 cache miss rates over time for SPEC CPU2000 benchmarks**