ABSTRACT

LIU, FANG. Analytically Modeling the Memory Hierarchy Performance of Modern Processor Systems. (Under the direction of Dr. Yan Solihin.)

In contemporary high performance microprocessor systems, memory hierarchy performance constitutes a crucial component of the overall performance, because it bridges the increasing gap between the CPU speed and DRAM main memory speed. More importantly, the expensive resources in the memory hierarchy, such as the last level cache and the off-chip memory bandwidth, are temporally and spatially shared among processes/threads in a multiprogramming or multitasking environment. Such resource sharing can cause severe resource contention, manifesting as performance volatility and system performance degradation.

Temporal sharing of memory hierarchy occurs through context switching, a technique allows multiple threads of execution to time-share a limited number of processors. While very useful, context switching can introduce high performance overheads, with one of the primary reasons being the cache perturbation effect. Between the time a thread is switched out and when it resumes execution, parts of its working set in the cache may be perturbed by other interfering threads, leading to (context switch) cache misses to recover from the perturbation.

The first goal of this dissertation is to understand how cache parameters and application behavior influence the number of context switch misses the application suffers from. We characterize a previously-unreported type of context switch miss that occurs as the artifact of the interaction of cache replacement policy and an application's temporal reuse behavior. We characterize the behavior of these "reordered misses" for various applications, cache sizes, and various amount of cache perturbation. As a second contribution, we develop an analytical model that reveals the mathematical relationship between cache design parameters, an application's temporal reuse pattern, and the number of context switch misses the application suffers from. We validate the model against simulation studies and find it is sufficiently accurate in predicting the trends of context switch misses with regard to cache perturbation amount.

The mathematical relationship provided by the model allows us to derive insights into precisely why some applications are more vulnerable to context switch misses than others. Through a case study on prefetching, we find that prefetching tends to aggravate context switch misses, and a less aggressive prefetching technique can reduce the number of context switch misses. We also investigate how cache size affects context switch misses. Our study shows that under relatively heavy workloads in the system, the worst case number of context switch misses for an application tends to increase proportionally with cache size, to an extent that may completely negate the reduction in other types of cache misses.

Spatial sharing of memory hierarchy resources is a common phenomenon in recent main-

stream computing platforms – Chip Multi-Processor (CMP) architectures. Recent CMPs allow multiple threads to execute simultaneously with each running on a single core, but multiple cores share the last level cache and off-chip pin bandwidth. It has been observed that such spatial sharing inevitably leads to single thread performance volatility, as well as overall system performance degradation. To alleviate the performance issue, last level cache and off-chip bandwidth partitioning schemes have been proposed in prior studies. While the effect of cache partitioning on system performance is well understood, little is understood regarding how bandwidth partitioning affects system performance, and how bandwidth and cache partitioning interact with one another.

The second goal of this dissertation is to propose a simple yet powerful analytical model that gives us the ability to answer several important questions: (1) How does off-chip bandwidth partitioning improve system performance? (2) In what situations is the performance improvement high or low, and what factors determine that? (3) In what way does cache and bandwidth partitioning interact, and is the interaction negative or positive? (4) Can a theoretically optimum bandwidth partition be derived, and if so, what factors affect it? We believe understanding the answers to these questions is very valuable to CMP system designers in coming up with strategies to deal with the scarcity of off-chip bandwidth in future CMPs.

Modern high performance processors widely employ hardware prefetching techniques to hide long memory access latency. While very useful, hardware prefetching tends to aggravate the *bandwidth wall* due to high bandwidth consumption, where system performance is increasingly limited by the availability of off-chip pin bandwidth in CMPs. Prior studies have proposed to improve prefetching policy or partitioning bandwidth among cores to improve bandwidth usage. However, they either study techniques in isolation, leaving the significant interaction unexplored, or perform them in an ad-hoc manner, resulting in important insights missed.

The third goal of this dissertation is to investigate how hardware prefetching and memory bandwidth partitioning impact CMP system performance and how they interact. To arrive at the goal, we propose an analytical model-based study. The model includes a composite prefetching metric that can help determine under which conditions prefetching can improve system performance, a bandwidth partitioning model that takes into account prefetching effects, and a derivation of the weighted speedup-optimum bandwidth partition sizes for different cores. Through model-driven case studies, we find several interesting observations that can be valuable for future CMP system design and optimization. We also explore simulation-based empirical evaluation to validate the observations and show that maximum system performance can be achieved by selective prefetching, guided by the composite prefetching metric, coupled with dynamic bandwidth partitioning.

Analytically Modeling the Memory Hierarchy Performance of Modern Processor Systems

by
Fang Liu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Gregory T. Byrd	Dr. James M. Tuck
Dr. Xiaosong Ma	Dr. Yan Solihin Chair of Advisory Committee

DEDICATION

To God, from whom all blessings flow, and to my parents, Jiaxing Liu and Laxiang Sun.

BIOGRAPHY

Fang Liu was born in a small town named Hanchuan in Hubei Province, P.R. China, on September 19th, 1984. She was enrolled in the Department of Information Electronics at Huazhong University of Science and Technology (HUST) in September 2002. She graduated from the prestigious Advanced Class and received her Bachelor of Engineering degree in June 2006. Since August 2006, she has been a PhD candidate under the supervision of Professor Yan Solihin in the Department of Electrical and Computer Engineering at North Carolina State University. She did an internship at IBM TJ Watson Research Center in the summer of 2009.

ACKNOWLEDGEMENTS

Firstly, I would like to express my deep gratitude to my research advisor Dr. Yan Solihin. This dissertation could not have been possible without his invaluable guidance and support. I have learned a lot of things about how to become a successful researcher in the past five years, including how to effectively and efficiently conduct research, how to prepare oral presentations, and how to write technical research paper. Besides the academic career, Dr. Solihin is also put his concern and time in inspiring his students' attitude and potential. I would like to particularly thank his encouraging and supporting in my qualifying exam.

I am also graceful of the important feedback and advice from my advisory committee members, Dr. Gregory Byrd, Dr. James Tuck and Dr. Xiaosong Ma. I especially appreciate Dr. Byrd's suggestions for my preliminary exam report.

I would like to thank all graduate students in ARPERS research group for their help and friendship. My appreciation especially goes to Seongbeom Kim and Abdulaziz Eker, for their guidance and contribution in starting up my first research project. I am also thankful to Fei Guo and Xiaowei Jiang, who have given me technical help in my research. I would also like to thank Siddhartha Chhabra, who has been the amazing partner for my course projects.

I would like to offer my inmost thank to my parents, Jiaxing Liu and Laxiang Sun, for their genuine and endless love for me. They have sacrificed a lot to raise me and support me, which motivated me to focus on my study. I would like to dedicate this dissertation for them.

Lastly, it is God's love that made it all happened.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

With the continuous development in integrated-circuit technology, Moore's law has predicted that the number of transistors on a CMOS silicon chip can be doubled every eighteen months. In the early stage of a modern processor design cycle, this trend, however, poses growing challenges in modeling and estimating the performance of the processor system.

To understand and estimate performance of processor systems, architects typically build low-level detailed simulation models, and run numerous simulations to evaluate the performance and obtain insights and design trade-offs. One of the most popular classes of simulation models is cycle-accurate simulators, such as SimpleScalar [15], Simics [60], and SESC [38]. The benefit of such simulation-based performance modeling is that it can be quickly adapted to new architectures and workloads. However, two major drawbacks of simulation models cannot be avoided. One is that creating and validating simulators themselves is time consuming. Even worse, the semiconductor development trend tends to increase the complexity of cycle-accurate simulators, because a growing number of resources are integrated on ever larger chips. The other drawback is that running benchmarks in the cycle-accurate simulators is extremely slow. It has been reported that simulators of out-of-order processors run programs thousands of times slower than the actual hardware [75]. With increasingly complicated architectures and workloads in the future, simulation time will keep growing [89].

An alternative performance modeling method is to deploy high-level analytical models [44]. An analytical model expresses the relationships among architecture parameters and application characteristics with mathematical formulae. While it may require collecting certain parameters in advance, this can usually be done via sampling or profiling based simulations, that take much shorter time to run than cycle-accurate simulations. In addition, the process of collecting parameters is a one-time cost that can be amortized over many cases. Once the model is built, it is very fast to tune the parameters and obtain the predicted performance results. Therefore, analytical performance modeling can significantly improve the efficiency of the architecture

design process, because less simulation time is required to obtain the same observations and design trade-offs. Another inherent benefit of analytical models is that they can reveal non-obvious trends and insights that are difficult to obtain by using cycle-accurate simulations, since the relationships of all variables are expressed in the mathematical formulae. Therefore, architecture design and evaluation can be improved by analytical modeling in terms of quality, as deeper insights can be obtained.

Among all performance aspects in modern processor design, memory hierarchy performance plays a significant role on overall performance, because it bridges the increasing gap between the CPU speed and DRAM main memory speed. More importantly, the expensive resources in the memory hierarchy, such as the last level cache as well as the off-chip memory bandwidth, are temporally and/or spatially shared among processes/threads in a multiprogramming or multi-tasking environment. Such resource sharing can cause severe resource contention, manifesting as single process performance volatility, as well as overall system performance degradation. In this dissertation, we will deploy analytical performance modeling to investigate the performance issues due to resource sharing in the memory hierarchy.

Temporal sharing of the memory hierarchy occurs due to context switching in modern computer systems, which allows multiple threads of execution to time-share a limited number of processors. While very useful, context switching can introduce high performance overheads, primarily because it incurs cache perturbation. Between the time a thread is switched out and when it resumes execution, parts of its working set in the cache may be perturbed by other interfering threads, leading to (context switch) cache misses to recover from the perturbation. The first goal of this dissertation is to understand how cache parameters and application behavior influence the number of context switch misses the application suffers from.

Spatial sharing of memory hierarchy resources is common in recent mainstream computing platforms – Chip Multi-Processor (CMP) architectures. Recent CMPs allow multiple threads to execute simultaneously with each thread running on a single core, but multiple cores share the last level cache and off-chip pin bandwidth. It has been observed that such spatial sharing may lead to single thread performance volatility, as well as overall system performance degradation. To alleviate the performance issue, last level cache and off-chip bandwidth partitioning schemes have been proposed in prior studies. While the effects of cache partitioning on system performance are well understood, how bandwidth partitioning affects system performance, and how it interacts cache partitioning are not clear. The second goal of this dissertation is to analyze those unclear factors.

Modern high performance processors widely employ hardware prefetching technique to hide long memory access latency. While very useful, hardware prefetching tends to aggravate the *bandwidth wall* due to high bandwidth consumption, where system performance is increasingly limited by the availability of off-chip pin bandwidth in CMPs. Prior studies have proposed

to improve prefetching policy or partitioning bandwidth among cores to improve bandwidth usage. However, they either study techniques in isolation, leaving the significant interaction unexplored, or perform them in an ad-hoc manner, resulting in important insights missed. The third goal of this dissertation is to investigate how hardware prefetching and memory bandwidth partitioning impact CMP system performance and how they interact.

1.1 Behavior and Implications of Context Switch Misses

One of the essential features in modern computer systems is context switching. It allows a system to provide an illusion of many threads of execution running simultaneously on a limited number of processors by time-sharing the processors. While very useful, context switching introduces high overheads *directly* and *indirectly* [4, 16, 22, 52, 63, 87, 88]. Direct context switch overheads include saving and restoring processor registers, flushing the processor pipeline, and executing the Operating System (OS) scheduler. Indirect context switch overheads include the perturbation of the cache and TLB states. When a process/thread is switched out, another process/thread runs and brings its own working set to the cache. When the switched-out process/thread resumes execution, it has to refill the cache and TLB to restore the state lost due to the context switch. Prior research has shown that indirect context switch overheads, mainly the cache perturbation effect, are significantly larger than direct overheads [16, 22, 52, 87]. For example, experimental results in [52] show that the indirect context switch cost is up to 262 times more than the average direct context switch cost on an IBM eSever with dual 2.0GHz Intel Pentium Xeon CPUs. We refer to the extra cache misses that occur as a result of the cache perturbation by context switching as *context switch cache misses*.

The number of context switch misses suffered by a thread is determined by the *frequency* of context switches as well as the number of context switch misses that occur after each context switch. The factors that affect the frequency of context switches and how they affect it are relatively easy to pinpoint and straightforward to understand. For example, the frequency of context switches is proportional to the level of *processor sharing*, i.e. the number of threads/processes that time-share a single processor. Factors that tend to increase the degree of processor sharing include thread-level parallelism and virtualization. Factors that tend to decrease the degree of processor sharing include multi-core design, which unfortunately increases the degree of memory hierarchy resource sharing.

It is not difficult to understand that the frequency of context switches occurring in the system is inversely proportional to the time quantum size allocated by the OS. However, factors that affect the number of context switch misses after a single context switch are not well understood. For example, it seems reasonable to expect that the part of the working set of a thread displaced from the cache due to a context switch will need to be fully reloaded back into the cache by the

thread through cache misses, which we refer to as *required reload misses*. One may expect that the number of context switch misses a thread suffers from to be closely related to the number of required reload misses. For the ease of discussion, we refer to this expectation as the *required reload misses hypothesis*. Unfortunately, the hypothesis is grossly inadequate at explaining the actual behavior of context switch misses. Figure 1.1 shows the average number of context switch misses that occur on a single context switch for eleven SPEC2006 benchmarks [82]. Each benchmark time-shares a single processor with *libquantum* using 5ms time quantum. The context switch misses are collected on a full system simulation that runs an unmodified Linux operating system (OS), with the processor having a 1MB L2 cache. [1] The x-axis shows the applications, sorted by L2 cache miss rate (shown in the parentheses) of each application when it runs alone on a dedicated processor.

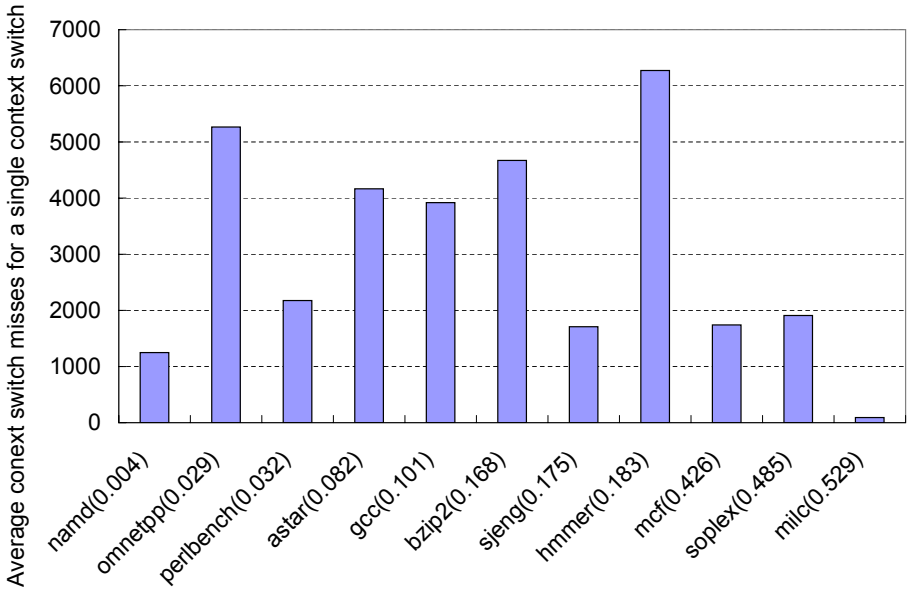


Figure 1.1: The average number of context switch misses for a single context switch suffered by SPEC2006 applications running on a processor with a 1MB L2 cache when they are so-scheduled with *libquantum*.

The required reload misses hypothesis cannot explain the variation of context switch misses across benchmarks, since the number of bytes in the working set displaced by *libquantum* on a single context switch is roughly the same across all benchmarks. The hypothesis ignores the fact that not all displaced bytes of the working set of a thread are going to be accessed again by the thread when it resumes execution. Hence, the temporal reuse pattern of an application

[1] Please refer to Section 2.1 for details on other processor and memory hierarchy parameters.

affects its context switch misses, but the exact relationship between them is still unclear. For example, ignoring cold misses, the cache miss rate of an application represents how likely a block that was used in the past and has been replaced from the cache will be reused. However, the figure shows that there is no apparent correlation between an application's miss rate with how many context switch misses it suffers from. For example, despite having similar miss rates, *hmmer* suffers from more than three times the context switch misses that *sjeng* suffers from.

Another reason why the required reload misses hypothesis is inadequate is that it assumes that context switch misses only arise due to the displaced working set. However, cache perturbation actually affects context switch misses through another channel. Specifically, it causes the non-displaced part of the working set to become less "fresh" in the cache, as it is shifted in its recency (or stack) order to be closer to the *least recently used* (LRU) position in the cache. This recency reordering causes the reordered cache blocks to have an elevated chance to be replaced by the *thread itself* when it resumes execution, before the thread has a chance to reuse them. Because these blocks are replaced by the thread itself (rather than by interfering threads from other applications), these misses have not been correctly reported as context switch misses in prior studies [1, 4, 48, 86], causing the number of context switch misses to be systematically under-estimated. To give an illustration of the magnitude of the under-estimation, Figure 1.2 shows the fractions of the two types of context switch misses for twelve SPEC CPU2006 applications. *Replaced misses* are context switch misses due to the part of working set that is displaced by the interfering thread, while *reordered misses* are ones due to the part of working set that is merely reordered in the cache. The evaluation setup is the same as in Figure 1.1, except that the result shown for each application is the average context switch misses when the application is co-scheduled with eleven other applications. The figure shows that reordered misses account for between 10% to 28% of the total context switch misses for a 1-MB cache and slightly higher for a 2-MB cache. In other words, not counting reordered misses as context switch misses may under-estimate the number of context switch misses rather significantly.

Clearly, a simple hypothesis such as the required reload misses hypothesis is inadequate in explaining what factors affect the number of context switch misses a thread suffers from and how they exactly affect it. Hence, the goal of this study is to *understand how cache parameters, a thread's temporal reuse patterns, and the amount of cache perturbation influence the number of context switch misses the thread experiences*. We hope that a better understanding of the nature of context switch misses can help researchers in coming up with techniques to reduce them. Our first contribution is characterizing context switch misses, especially focusing on the reordered misses: their magnitude, how they affect different applications, and how they are affected by cache sizes and the amount of cache perturbation. The main findings are: (1) context switch misses can contribute to a significant increase in the total L2 cache misses; (2) they tend to increase along with the cache size up until the cache size is large enough to hold the

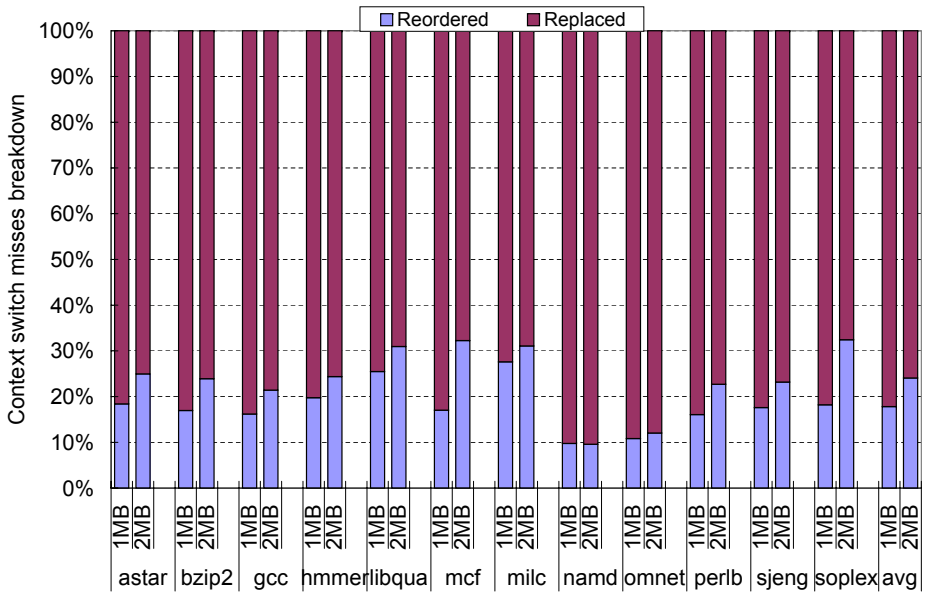


Figure 1.2: Two types of context switch misses suffered by SPEC2006 applications running on a processor with 1-MB L2 cache or 2-MB cache.

entire combined working sets; (3) reordered misses tend to contribute to an increasing fraction of context switch misses as the cache size increases; and (4) the maximum number of reordered misses occurs when cache perturbation displaces roughly half of the total cache blocks.

Our second contribution is an analytical model-based investigation to establish how the temporal reuse pattern of a thread, the amount of cache perturbation, and the number of context switches are *mathematically related*. Compared to empirical studies, the mathematical relationship allows us to gain clearer insights into the behavior of context switch misses. We validate our Markov-based model against the simulation results of SPEC2006 applications. We find that the model is sufficiently accurate in predicting the trends of context switch misses. The model allows us to derive insights into precisely why some applications are more vulnerable to context switch misses than others. Then, as one case study, we apply the model to analyze how prefetching and context switch misses interact with each other in various applications. The investigation leads us to a previously-unreported observation that prefetching can aggravate the number of context switch misses for an application. In addition, perversely, the more effective prefetching is for an application, the higher the number of context switch misses the application suffers from. We also investigate how cache sizes affect the number of context switch misses. Our study shows that under relatively heavy workloads in the system, the worst-case number of context switch misses for an application tends to increases with cache sizes, to an extent that may completely negate the reduction in other types of cache misses.

1.2 Effects of Bandwidth Partitioning on CMP Performance

Chip Multi-Processors (CMPs) have recently become a mainstream computing platform. Recent CMP designs allow multiple processor cores to share expensive resources, such as the last level cache and off-chip pin bandwidth. Such sharing has produced an interesting contention-related performance phenomena, in which the system throughput, as well as individual thread performance, are highly volatile, depending on the mix of applications that share the resources and on how the resources are partitioned among the applications. To improve system performance and/or reduce the performance volatility of individual threads, researchers have proposed to either partition the last level cache [13, 14, 26, 30, 36, 41, 54, 68, 70, 71, 77, 83, 84] or partition off-chip bandwidth [33, 50, 64, 65, 66, 72] usage between cores, with partition sizes chosen to optimize a particular goal, such as maximizing throughput or fairness. The growing number of cores on a chip increases the degree of sharing of the last level cache and off-chip bandwidth, making it more important to reach an optimum partition for each resource.

THe effects of bandwidth partitioning on system performance are not as well understood as the effects of cache partitioning. For example, it is well known that cache partitioning can reduce the total number of cache misses by reallocating cache capacity from threads that do not need much cache capacity to those that do, and the reduction in the total number of cache misses improves the overall system performance. However, allocating fractions of off-chip bandwidth to different cores does not reduce the total number of cache misses. Instead, it only prioritizes the off-chip memory requests of one thread over others. Hence, while bandwidth partitioning can clearly affect an individual thread's performance, the aspects of how off-chip bandwidth partitioning affects system performance are not yet well understood.

In addition, despite the wealth of studies in last level cache and off-chip bandwidth partitioning, the interactions between cache partitioning and bandwidth partitioning are not well understood. Most studies in cache partitioning ignore bandwidth partitioning [13, 14, 26, 30, 36, 54, 68, 70, 71, 77, 83, 84], while most studies in bandwidth partitioning assume private caches or a statically-partitioned shared cache [50, 64, 65, 66, 72]. Only recently coordinated cache partitioning and bandwidth partitioning were explored together using an artificial neural network (ANN) optimizer, where it was shown to outperform the cases in which cache or bandwidth partitioning were applied individually [8]. However, since ANN is a black box optimizer, the nature of interaction between cache and bandwidth partitioning remains unexplored.

One theory that has been mentioned in the literature is that bandwidth partitioning's impact on performance is *secondary* to that of cache partitioning [12, 84]. The theory certainly makes sense, because cache partitioning can significantly reduce the total number of cache misses of applications that share the cache, resulting in reduction in the total off-chip memory traffic; in contrast, bandwidth partitioning cannot reduce total off-chip memory traffic, since it

merely prioritizes cache misses or write back requests from some cores over others. Hence, it is intuitive to guess that the performance impact of bandwidth partitioning is likely secondary to that of cache partitioning. Taking it one step further, one may further hypothesize that when cache partitioning is employed, bandwidth partitioning may be unnecessary because it has little additional performance benefit compared to that of cache partitioning. This hypothesis also makes sense because when off-chip bandwidth is plentiful, bandwidth partitioning is unnecessary, while cache partitioning can still improve performance because of the reduction in the number of cache misses. When off-chip bandwidth is scarce, bandwidth partitioning cannot alleviate the scarcity because it does not reduce the number of memory requests, while cache partitioning can.

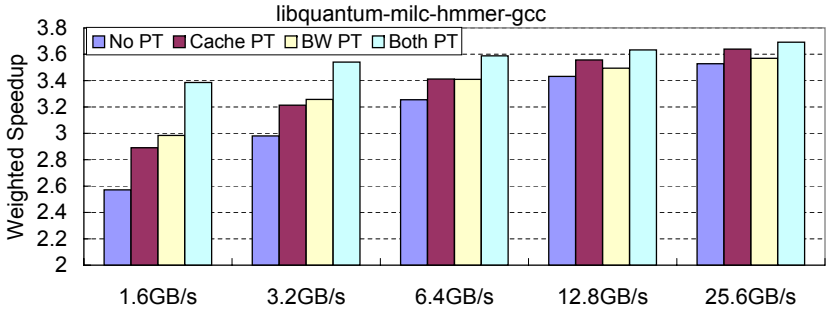


Figure 1.3: Weighted speedup of four applications running on a 4-core CMP with a 2MB shared L2 cache, assuming various peak off-chip bandwidth.

Interestingly, however, our empirical evaluation shows that the hypotheses are not necessarily correct. Figure 1.3 shows the weighted speedup [3] of four sequential SPEC2006 applications (*libquantum*, *milc*, *hmmer*, and *gcc*) running on a 4-core CMP system with a 2MB shared L2 cache, with different available peak off-chip bandwidth. The four bars for each peak bandwidth correspond to when no partitioning is applied, only cache partitioning is applied, only bandwidth partitioning is applied, and both cache and bandwidth partitioning are applied. [2] The figure shows that (1) bandwidth partitioning can improve the overall system performance, (2) it even outperforms cache partitioning in some cases (1.6GB/s and 3.2GB/s), and (3) combined cache and bandwidth partitioning slightly outperforms the product of speedup ratios of individual partitioning techniques, implying that there sometimes can be a synergistic interaction between them.

These surprising results demand explanations that are beyond what the hypotheses above

[2]The cache partitioning scheme attempts to minimize the total number of cache misses, and the bandwidth partitioning scheme attempts to maximize the weighted speedup of applications. Details of the partitioning algorithms and simulation parameters can be found in Section 3.3.1.

can provide. Hence, *the goal of this study is to understand what factors contribute to the performance impact of bandwidth partitioning, and how they fundamentally interact with cache partitioning.* To arrive at the goal, we propose an analytical model that is simple yet powerful enough to model the impact of bandwidth partitioning on performance. Our analytical model is built on top of the additive Cycle Per Instruction (CPI) model [20, 59, 78] and queuing theory [55], taking various system parameters (CPU frequency, cache block size, and peak available bandwidth), as well as application cache behavior metrics (various CPIs, and miss frequency) as input. Studying the model, coupled with results from empirical evaluation, we arrive at several interesting findings, among them are:

- The reason why bandwidth partitioning can improve the overall system performance is that the impact of queuing delay on memory requests due to bandwidth contention affects different applications to different degrees. By favoring memory requests from applications which are more sensitive to queuing delay, the overall system performance can be improved.

- *Whether or not* system performance can be improved by bandwidth partitioning depends on the difference between miss frequencies (misses per unit time) among different threads. However, the *magnitude* of system performance improvement due to bandwidth partitioning is highly dependent on the difference between peak available bandwidth and the rate at which off-chip memory requests are generated. Our model explains why this is so.

- Deploying cache partitioning that minimizes the total number of cache misses as a goal may in some cases decrease the impact of bandwidth partitioning on system performance, while in other cases, it may increase the impact of bandwidth partitioning on system performance. Thus, the speedup from employing both cache and bandwidth partitioning can sometimes exceed the multiplication of speedup ratios resulting from each partitioning technique applied individually. Our model explains the reasons for the occurrence of both cases, and shows under what situations each case occurs.

- Our model allows the derivation of the bandwidth partition sizes that optimize weighted speedup. While impractical to implement, we show that the optimal bandwidth can be approximated, and our algorithm that is based on this approximation outperforms other partitions as well as equal (fair) partitioning.

The findings in our study carry significant implications for CMP system design. In the future, the number of cores on a chip will continue to double every 18-24 months according to Moore's law, whereas the off-chip bandwidth is projected to grow only 10-15% per year according to ITRS Roadmap [35]. Consequently, the off-chip bandwidth available per core

will become increasingly scarcer and contended by the cores [74], increasing the importance of bandwidth partitioning.

1.3 Impact of Hardware Prefetching and Bandwidth Partitioning

Several decades of the persistent gap between microprocessor and DRAM main memory speed improvement has made it imperative for today's microprocessors to hide hundreds of processor clock cycles in memory access latency. In most high performance microprocessors today, this is achieved by employing several levels of caches augmented with *hardware prefetching* [7, 24, 29, 32]. Hardware prefetching works by predicting cache blocks that are likely needed by the processor in the near future, and fetching them into the cache early, allowing processor's memory accesses to hit (i.e. find data) in the cache. Since prefetching relies on prediction of future memory accesses, it is not 100% accurate in its prediction, implying that the latency hiding benefit of prefetching causes an increase in off-chip memory bandwidth usage and cache pollution.

In single core processor systems, for most workloads, off-chip bandwidth was rarely saturated from the use of prefetching, making prefetching almost always beneficial. However, the shift to multi-core design has significantly altered the situation. Moore's Law continues to allow the doubling of the number of transistors (and hence number of cores) every approximately 2 years, increasing the off-chip bandwidth pressure at Moore's Law speed. However, the availability of off-chip bandwidth is only projected to grow at a much lower 15% annual rate, due to the limitations in pin density and power consumption [35]. This discrepancy leads to a problem called the *bandwidth wall*, where system performance is increasingly limited by the availability of off-chip bandwidth [21, 39, 40, 41, 42, 43, 56, 57, 74]. Hence, the demand for efficient use of off-chip bandwidth is tremendous and increasing.

At least two methods have been recently proposed to improve the efficiency of off-chip bandwidth usage. One method is to improve prefetching policies. To reduce the performance damage from the increase in bandwidth usage in prefetching, some studies have proposed throttling or eliminating the useless/bad prefetches from consuming bandwidth [18, 19, 67, 79, 81, 90], and tweaking the memory scheduling policy to prioritize demand and profitable prefetch requests [18, 50, 51]. Another method is to partition the off-chip bandwidth usage among cores [8, 33, 57, 64, 65, 66, 72, 80], with partition sizes chosen to optimize a particular goal, such as to maximize throughput or fairness. However, these studies suffer from several drawbacks. First, the studies address one technique but ignore the other: prefetching studies do not include bandwidth partitioning, whereas bandwidth partitioning studies assume systems that have no prefetching. As a result, the significant interaction between them was missed,

and the opportunity for these techniques to work in synergy was left unexplored. Second, the studies were performed in an ad-hoc manner, yielding performance improvement but missing important insights.

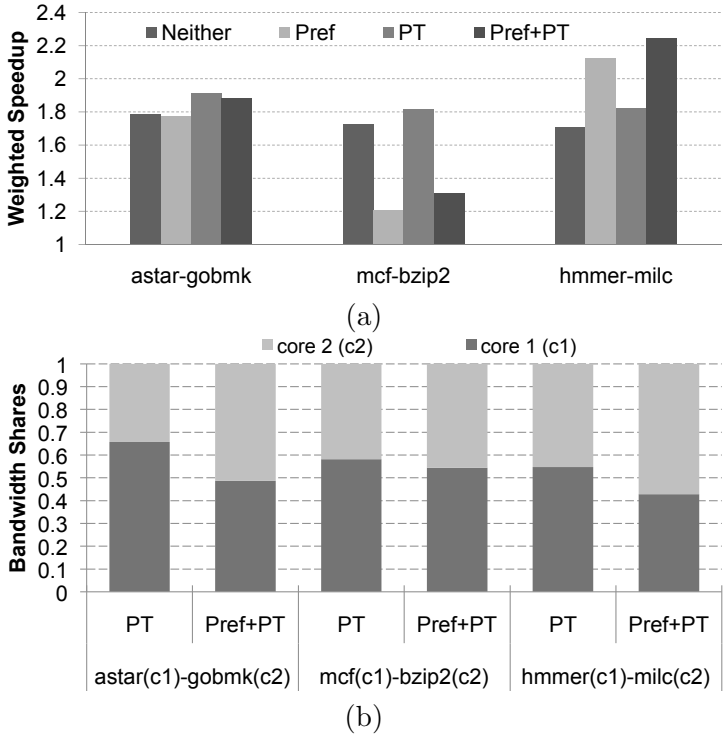


Figure 1.4: Weighted speedup (a) and optimum bandwidth allocation (b) for co-schedules running on a dual-core CMP.

To demonstrate the need for exploring prefetching and bandwidth partitioning jointly, Figure 1.4(a) shows how prefetching and bandwidth partitioning can affect each other. In the figure, system performance measured as weighted speedup [3] (referring to Equation 3.1) of three pairs of SPEC2006 [82] benchmarks running as a co-schedule on a dual-core CMP system is shown with four configurations: base system with no prefetching or bandwidth partitioning (Neither), hardware prefetching only (Pref), bandwidth partitioning only (PT), and both prefetching and bandwidth partitioning (Pref+PT). [3] The three co-schedules highlight different interaction cases. For *hmmer-milc*, prefetching improves performance and bandwidth partitioning improves it further due to offsetting the effect of the prefetcher's increase in off-chip bandwidth usage. For *astar-gobmk* and *mcf-bzip2*, prefetching hurts performance due to

[3] A stream prefetcher similar to previous studies [45, 50] is used. Optimum bandwidth partitioning scheme from [57] is used. More details can be found in Section 4.3.1.

high off-chip bandwidth consumption. For *mcf-bzip2*, applying bandwidth partitioning is not the right solution, since it cannot recover the lost performance due to prefetching (i.e. Pref+PT < Neither). Such a conclusion cannot be derived in prior studies, when only bandwidth partitioning was studied [8, 33, 57, 64, 65, 66, 72, 80], or when prefetchers were always turned on [18, 50, 51]. The figure points out the need to understand when and under what situations the prefetcher of each core should be turned on or off, and how bandwidth partitioning should be implemented in order to optimize system performance.

Performing the studies in an ad-hoc manner often misses important insights. For example, it has been assumed that a core that enjoys useful prefetches should be rewarded with a higher bandwidth allocation, whereas a core for which prefetching is less useful should be constrained with a lower bandwidth allocation [18]. However, our experiments show the opposite. Maximum weighted speedup is achieved when a core with highly useful prefetching is given less bandwidth allocation, and the resulting excess bandwidth is given to cores with less useful prefetching. Figure 1.4(b) shows the bandwidth allocation for each core that maximizes the system throughput. The application that has more useful prefetching (higher prefetching coverage and accuracy) runs on Core 1. Comparing the two bars for each co-schedule, it is clear that in order to maximize system throughput, the applications that show less useful prefetching should receive higher bandwidth allocations.

The goal of this study is to *understand the factors that contribute to how prefetching and bandwidth partitioning affect CMP system performance, and how they interact.* We propose an analytical model-based study to address those issues. Studying the model, coupled with empirical evaluation, we arrive at several interesting findings, among them are:

- Deploying prefetching makes the available bandwidth scarcer and therefore increases the effectiveness of bandwidth partitioning in improving system performance.
- The decision of turning prefetchers on or off should be made prior to employing bandwidth partitioning, because the performance loss due to prefetching in bandwidth-constrained systems cannot be fully repaired by bandwidth partitioning. In deciding whether to turn on or off the prefetcher for each core, traditional metrics such as coverage and accuracy are insufficient. Instead, we discover a new metric that works well for this purpose.
- The theoretical optimum bandwidth allocations can be derived, and a simplified version that is implementable in hardware can approximate it quite well.
- The conventional wisdom that rewards a core that has more useful prefetching with a larger bandwidth allocation is incorrect when prefetchers of all cores are turned on. Instead, its bandwidth allocation should be slightly more constrained. Our model explains why this is so.

1.4 Organization of the Dissertation

The rest of the report is organized as follows. Chapter 2 characterizes the behavior of context switch misses with a focus on reordered misses, and then proposes an analytical model to reveal the mathematical relationship between cache parameters, the temporal reuse behavior of a thread, and the number of context switch misses the thread suffers from. Chapter 3 presents a simple yet powerful analytical model to understand what factors contribute to the performance impact of bandwidth partitioning and how they fundamentally interact with cache partitioning. Chapter 4 shows an analytical model-based study to investigates how hardware prefetching and memory bandwidth partitioning impact CMP system performance and how they interact with each other. Finally, Chapter 5 retrospects the three studies and provides the reflections and opinions based on this research.

Chapter 2

Behavior and Implications of Context Switch Misses

This chapter is organized as follows: Section 2.1 characterizes the context switch misses across different applications, Section 2.2 explains the analytical cache model in details, Section 2.3 analyzes the validation results, Section 2.4 applies the model on two case studies: prefetching and cache size respectively, Section 2.5 discusses the related work, and Section 2.6 concludes this chapter [21, 56].

2.1 Characterizing Context Switch Misses

In this section, we will present the first contribution of this paper on characterizing the behavior of reordered misses for various applications, cache sizes, and degrees of cache perturbation.

2.1.1 Types of Context Switch Misses

Figure 2.1 illustrates the causes and types of context switch misses. The figure shows a single cache set in a 4-way set associative cache. Figure 2.1(a) shows the cache state right before a process is context switched, containing blocks A, B, C, and D in use-recency order with A being the *most-recently used* (MRU) block while D being the *least-recently used* (LRU) block. When the process resumes execution after the context switch, the cache has been perturbed by another process which left one of its cache blocks at the MRU position (indicated by the "hole" in Figure 2.1(b)). The cache perturbation causes block D to be replaced from (or shifted out of) the cache. Suppose that now the application chronologically accesses block D, C, and then E. Without the cache perturbation, only the access to block E would result in a cache miss. With the cache perturbation, the access to D results in a cache miss (Figure 2.1(c) shows the resulting cache state). Since D was *replaced* by the perturbation, we refer to the miss as

a *replaced context switch miss*. Next, when the process accesses block C, it also suffers from a cache miss (Figure 2.1(d) shows the resulting cache state). Note that block C was merely reordered in the LRU stack and was not replaced by the cache perturbation, so the fact that it causes a cache miss is the artifact of the LRU cache replacement policy. We refer to such a miss as a *reordered context switch miss*. Finally, an access to E results in a cache miss, but it is not a context switch miss since it occurs regardless of whether there was a context switch.

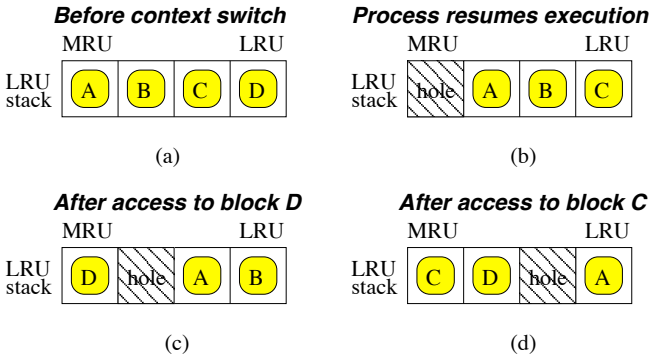


Figure 2.1: The content of a cache set right before a process is context switched (a), when it resumes execution (b), after an access to block D (c), and after an access to block C (d).

From the figure, we can observe that the number of context switch misses after a context switch is influenced by the probability that replaced or reordered blocks will be accessed again when a thread resumes execution. Such probability is affected by the temporal reuse pattern of the application, and the specific causal relationship will be established in Section 2.2. In this section, we are interested in the composition of context switch misses: what fraction of them is due to reordered misses versus due to replaced misses. Clearly, if the context switch results in the entire cache state to be displaced from the cache, then all context switch misses will be replaced misses, since no block is reordered in the cache. Hence, reordered misses only occur when only a fraction of the cache is displaced during a context switch.

2.1.2 Characterization Methodology

We use a cycle-accurate processor simulator based on Simics [60], a full-system processor simulation infrastructure. We run an unmodified Fedora Core 4 distribution of the Linux Operating System on the processor. The processor has a scalar in-order issue pipeline with a 4GHz frequency. The memory hierarchy has two cache levels. The L1 instruction cache has a 16KB size, 2-way associativity. The L1 data cache has a 32KB size and 4-way associativity. The L2 cache is 8-way associative, and its cache size ranges from 512KB to 4MB. In this study, we focus

on characterizing the behavior of L2 cache misses instead of system performance expressed as execution time; hence the latency to access each cache does not matter. All caches have a block size of 64 bytes, implement write-back policy and use the LRU replacement policy. To collect the number of context switch misses, we implement two L2 caches: one *real* L2 cache that is affected by the cache perturbation, and a hypothetical *isolated* L2 cache that is not affected by the cache perturbation. Each L1 cache miss is passed to both caches to determine whether it will hit or miss in each of the L2 caches. The difference in the numbers of cache misses between these two caches measures the number of L2 context switch misses.

Table 2.1: SPEC2006 benchmarks. The unit for L2 miss frequency is the average number of cache misses in a single cache set that occurs in 50 milliseconds. The L2 cache size is 512KB.

Benchmarks	Miss Rate	Miss Frequency (#misses/set/50ms)
astar	10.72%	181
bzip2	27.35%	224
gcc	14.15%	206
gobmk	3.23%	96
h264ref	0.66%	3
hmmer	33.38%	131
lbm	32.27%	465
libquantum	50.81%	269
mcf	46.64%	602
milc	56.26%	79
namd	0.48%	5
omnetpp	13.32%	94
perlbench	2.85%	98
povray	0.34%	5
sjeng	20.00%	206
soplex	54.76%	488
sphinx3	70.96%	515

We consider all seventeen C/C++ benchmarks from the SPEC2006 benchmark suite [82]. We omit benchmarks written in Fortran because of a limitation in our compiler infrastructure. Table 2.1 shows each benchmark's L2 cache miss rates (number of L2 cache misses divided by L2 cache accesses) and the L2 cache miss frequency (number of L2 cache misses per unit time) on a 512KB cache. Of the seventeen benchmarks, we choose twelve representative applications that have distinct temporal reuse patterns (including miss rates) and miss frequency. We compile the benchmarks using `gcc` compiler with `O1` optimization level into `x86` binaries. We use the *ref* input sets and simulate two billion instructions after skipping the initialization phase of each benchmark, which is identified through manual inspection of the source code.

2.1.3 Characterization Results

Figure 2.2 shows the breakdown of the L2 cache misses for twelve SPEC2006 benchmarks when each benchmark is co-scheduled with another interfering benchmark, for various cache sizes. Since there are eleven other benchmarks, there are eleven possible co-schedules. Hence, the number of L2 cache misses of each benchmark is taken as the average over eleven co-schedules the benchmarks can have, normalized to the case in which 512-KB cache is used. A time slice of 5ms is used for both benchmarks in a co-schedule. The Linux kernel version 2.6 assigns time quanta between 5 ms to 200 ms based on a process/thread priority level, so 5ms corresponds to the lowest priority level.

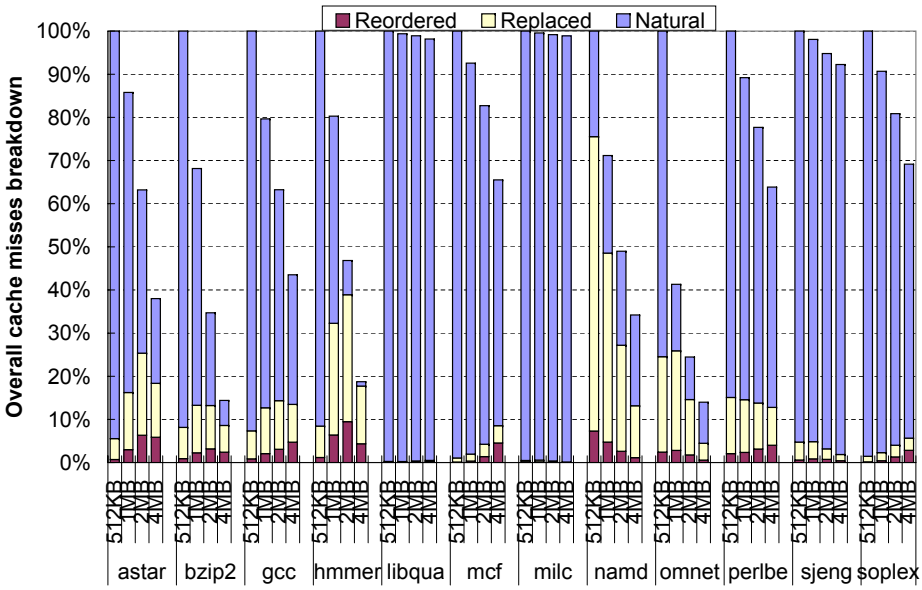


Figure 2.2: Breakdown of the types of L2 cache misses suffered by SPEC2006 applications with various cache sizes.

The figure shows that as expected, as the cache size increases, the "natural misses" (the sum of cold, capacity, and conflict misses) decrease. However, the number of context switch misses exhibits very different behavior. Sometimes, they decline (in namd, perlbench), increase (in mcf, soplex), or increase then decline (in astar, bzip2, gcc, hmmer, omnetpp and sjeng). When the cache is very small, there are not many cache blocks that can be displaced by a cache perturbation, so the number of context switch misses is low. As the cache becomes larger, a cache perturbation may displace more cache blocks, causing more context switch misses. However, when the cache becomes large enough to hold the total working sets of both

benchmarks, cache perturbation only displaces very few blocks, so the number of context switch misses declines. Rather than being displaced, most blocks are now reordered, but reordered blocks do not incur context switch misses if there are no other misses to cause them to be displaced when the thread resumes execution. Hence, the absolute number of reordered misses also declines as the cache becomes large enough.

The figure also shows that when both benchmarks in a co-schedule are assigned a time slice of 5ms, context switch misses can represent a significant fraction of the total L2 cache misses, especially on larger cache sizes, which implies that they need to be taken into account when designing a system. Obviously, if a larger time slice is used, the frequency of context switches occur in the system will decrease, and hence the total number of context switch misses will be reduced as well. Figure 2.3 shows the number of context switch misses each benchmark suffers from when it is assigned time slices of 5ms, 25ms and 50ms respectively, while the co-scheduled benchmarks still use 5ms. The results are normalized to 5ms case for each benchmark. The figure shows that as expected, when the time slice of the primary benchmark is increased to $r\times$, the number of context switch misses it suffers from decreases to $\frac{1}{r}$ of the original number of context switch misses on average (i.e.,compared to 5ms, approximately 20% for 25ms and 10% for 50ms).

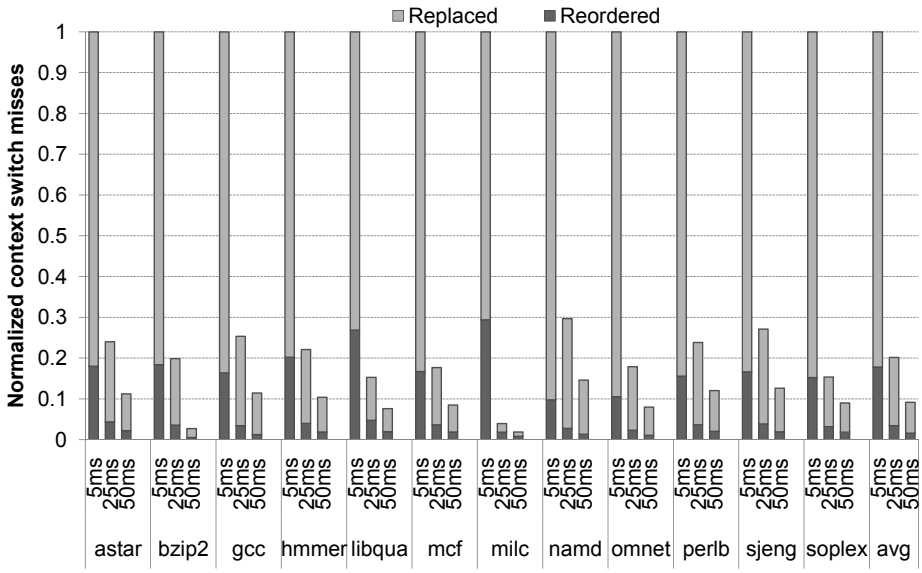


Figure 2.3: Normalized number of context switch misses for various time slices on an 8-way associative 1-MB L2 cache.

However, caution is needed before making a conclusion that context switch misses are easy

to eliminate through the use of large time slices, for several reasons. First, in many operating systems including Linux, applications with low priorities in the system are granted small time quanta, causing them to suffer from a large number of context switch misses. Secondly, some applications that are I/O intensive often block before they use up a time quantum. Thirdly, applications that time-share a processor with an I/O intensive application are also forced to context switch very frequently due to interrupts. For example, we observed that on typical machines, when the GNU compiler `gcc` and a secure copy application `scp` are co-scheduled, both suffer from a very frequent context switches and each of them runs for much less than 1ms before it is context switched out (versus 5ms we use in this study). Finally, perhaps the most important reason why a large time slice may not be a solution is that it becomes much less effective in reducing the number of context switch misses as the cache size increases. We will show this in Section 2.4.2.

How does the composition of replaced and reordered misses change with different cache sizes? Figure 2.4 shows the number of reordered vs. replaced misses as a fraction of the total context switch misses. When the cache size is small, a cache perturbation is more likely to displace cache blocks than reorder them, so the fraction of reordered misses is small (roughly 15% on average for a 512KB L2 cache). When the cache size increases, a cache perturbation is more likely to reorder cache blocks than to replace them, so the fraction of reordered misses increases (to roughly 32% on a 4MB L2 cache). In some applications, reordered misses even account for roughly 50% of all context switch misses (libquantum, mcf, and soplex in 4MB cases). From the figure, we observe that reordered misses are a significant component of context switch misses, and counting only replaced misses as context switch misses as in prior studies [1, 4, 48, 86] can result in significantly and systematically under-estimating the total number of context switch misses.

The number of context switch misses an application suffers from when it is co-scheduled with other applications is essentially determined by two factors: the amount of cache perturbation (how many blocks are reordered and replaced from the cache), and how likely the affected blocks will be needed again by the application. The amount of cache perturbation depends on how many interfering applications there are, how long they run, and how fast they bring in new cache blocks to cause the cache perturbation. Since it is difficult to produce all possible scenarios, in order to better understand the behavior of reordered misses, we abstract the amount of cache perturbation by a single metric which we call the *shift amount*. During a cache perturbation, when a new block is brought in, it is placed at the most recently used entry in a cache set, causing all other blocks in the set to be shifted in the recency order, with the LRU block displaced from the cache set. The shift amount refers to the number of new blocks brought into each cache set, which is also equal to the number of positions old blocks are shifted in their recency order. A larger shift amount indicates more cache perturbation from other applications.

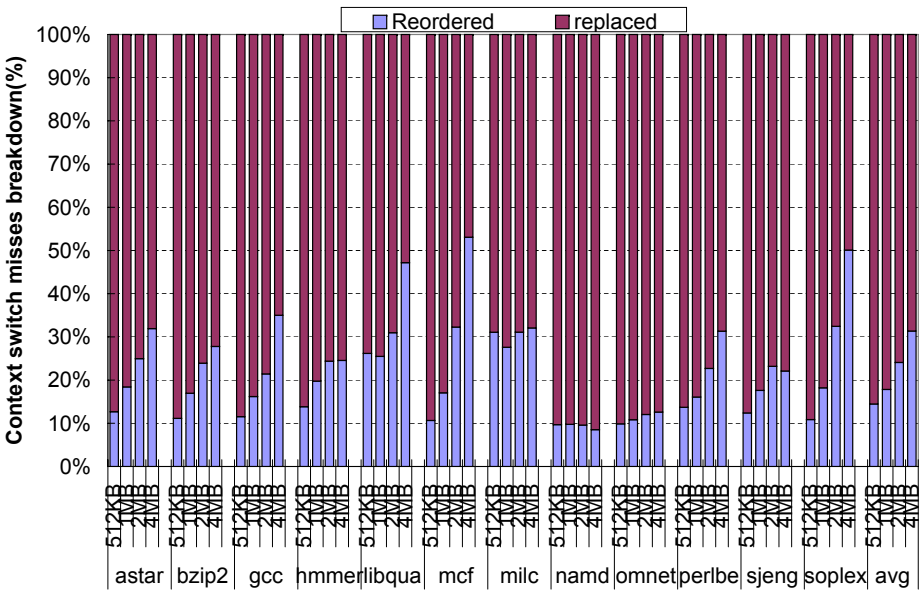


Figure 2.4: The composition of context switch misses for various cache sizes.

The maximum value of shift amount is by definition equal to the cache associativity.

With the amount of cache perturbation abstracted as the shift amount, we run each of the twelve SPEC2006 benchmark, and every 5ms, we pause the benchmark, perturb the L2 cache by the shift amount, and resume the benchmark execution. Figure 2.5 shows the percentage increase in the number of L2 cache misses due to context switching. From this figure, we can make a few observations. First, the number of context switch misses monotonically increases as the shift amount increases for all benchmarks. The reordered misses, however, reach their maximum when the shift amount is 4 (50% of the cache size) or 6 (75% of the cache size). The number of reordered misses is affected by the number of blocks reordered and how many recency order positions they are shifted. With a small shift amount, many blocks are reordered but they are only slightly shifted. As the shift amount grows, fewer blocks are reordered but they are shifted by more positions in the recency order, causing an increase in reordered misses. However, a very large shift amount implies that very few blocks are reordered (because more blocks are replaced), so the number of reordered misses declines.

The analysis explains why the fraction of reordered misses increases with larger cache sizes in Figure 2.4. With the time quantum unchanged, larger cache sizes correspond to going from a very large shift amount to a smaller shift amount, causing an increase in the fraction of reordered misses.

Overall, from this characterization study, we have learned that the number of context switch misses tends to increase as the cache size increases but only up to a point, after which it starts

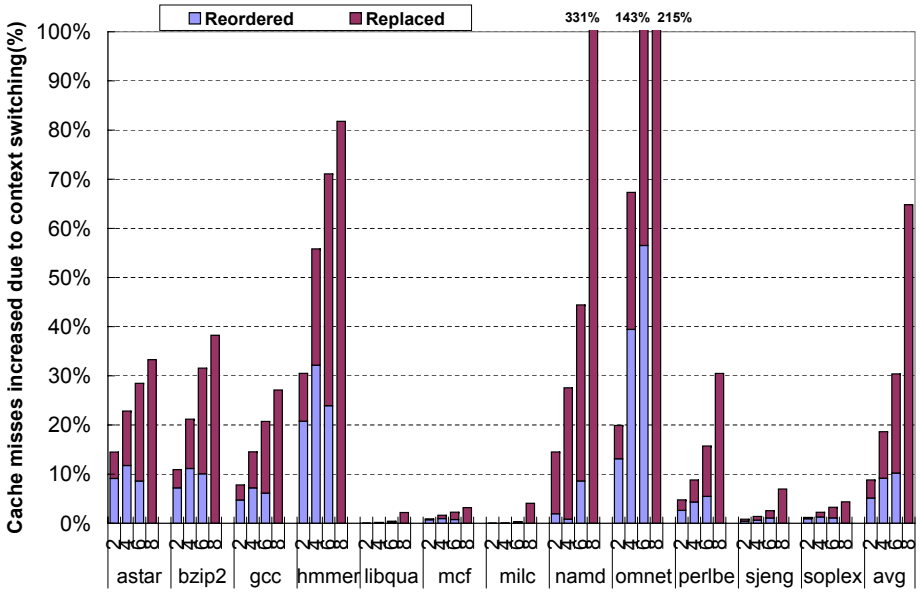


Figure 2.5: Increase in the L2 cache misses due to context switch misses for various shift amounts (2, 4, 6, and 8) on an 8-way associative 1-MB L2 cache. A shift amount of 8 means the entire cache state is replaced.

to decline. Context switch misses also correspond to an increasing fraction of total cache misses. The number of reordered misses tends to reach its peak when cache perturbation affects roughly a half of the cache size, while the fraction of reordered misses of the total context switch misses increases as the cache size increases. Cache designers must take into account context switch misses in addition to other types of misses, especially when the level of processor sharing is high (i.e. high cache perturbation), and when the cache size is relatively large.

Although simulation studies can help us understand major behavior trends of context switch misses, they are insufficient for understanding how exactly different applications suffer from different numbers of context switch misses. To gain such understanding, as a second contribution, in the next section we derive the mathematical relationship between an application's temporal reuse pattern and the behavior of context switch misses through an analytical Markov-based model.

2.2 The Analytical Cache Model

2.2.1 Assumptions and Input of the Model

Estimating how many context switch misses an application will likely suffer from requires us to determine the probability of an individual cache access incurring a context switch miss, and sum up such probabilities over a group of cache accesses. From the example in Figure 2.1,

estimating the probability of a single cache access incurring a context switch miss requires us to take into account: (1) the amount of cache perturbation introduced by the context switch, (2) the temporal reuse pattern of the application, (3) the cache replacement policy, and (4) and the location of the holes introduced by the context switch at the time the access is made. The location of the holes matters because reuses to cache blocks that are in more-recent stack positions than the holes do not change the location of the holes, while those that are in less-recent stack positions than the holes shift the position of the holes.

The temporal reuse pattern of an application can be captured through a popular profiling technique called *stack distance profiling* [10, 11, 25, 61], or sometimes also referred to as marginal gain counters [83]. Basically, stack distance profiling records the distribution of accesses to different LRU-stack positions, either as global information for the entire cache, or as local information for each cache set. To collect the global information, for an A-way associative cache, $A+1$ counters are kept: $C_1, C_2, \ldots, C_A, C_{>A}$. The LRU-stack positions are enumerated, with the MRU position as the first stack position, and the LRU position as the A^{th} position. On each cache access (to any set), one of the counters is incremented. A cache access to a block at the i^{th} LRU-stack position increments C_i. If a cache access does not find the cache block in the LRU stack, it increments $C_{>A}$. After the access, the stack is updated, with the block recently accessed moved up to the first stack position, while other blocks between the first stack position and the old position of the accessed block are shifted down. Collected over a long period, the stack distance profile can be used to estimate the probability that a block in the i^{th} stack position to be reused in the next cache access as:

$$P_{reuse}(i) = \frac{C_i}{C_{>A} + \sum_{j=1}^{A} C_j} \tag{2.1}$$

To collect the local (per-set) profiling information, we maintain one set of counters for each cache set. If cache accesses are uniformly distributed across all cache sets, the global stack distance profile is sufficient in capturing the overall temporal reuse pattern of the application. Due to its simplicity, we collect the global stack distance profile. A stack distance profile can be obtained statically through static analysis [11], or at run time through simulation or direct hardware implementation [83].

When a cache brings in a new block, it must find a block to replace to make room for the new block. Which block is selected for replacement is determined by the cache replacement policy. Applications often have regular *temporal reuse* behavior in which it tends to reuse more recently used data more than less recently used data. For this reason, most cache implementations today implement least-recently used (LRU) replacement policy or its approximations. Hence, in our model we assume LRU cache replacement policy.

Similar to the characterization in Section 2.1.3, we abstract the amount of cache perturba-

tion as the *shift amount*, i.e. the number of stack positions a process' data blocks are shifted in the LRU stack, which is also equivalent to the number of holes introduced in each cache set. We refer to such number as the *shift amount*, denoted as δ. The range of values for δ is $\delta = 0 \ldots A$.

We further assume that the location of the holes in all cache sets to be contiguous, which rests on two smaller assumptions. First, consecutive context switches are sufficiently separated in time that holes from one context switch have been completely shifted out prior to the next instance of context switch. The assumption is mostly valid given that time quanta used by Operating Systems today are in the order of at least a few milliseconds. Second, processes that time-share a processor do not share data, which is mostly valid for independent and unrelated processes. Communicating processes may have a high degree of data sharing, but context switching is not necessarily harmful for them, so we do not seek to model them. Finally, the assumption of contiguous holes is not a fundamental restriction of our model. Its purpose is to simplify the model by reducing the state space. The assumption can be relaxed at the expense of additional model complexity.

2.2.2 Model Formulation

We will start by defining some basic terms that we will use throughout the discussion. The process of which the context switch misses we want to model is referred to as the *target process*. The target process' miss rate without cache perturbation from context switches is referred to as the *natural cache miss rate*, computed as the ratio of the number of non-context switch misses divided by the number of cache accesses. To simplify the discussion, we begin our model formulation for a *single cache set* and a *single instance of context switch*. The cache state necessary for our model is defined as follows.

Definition 1 *For a single cache set in an A-way associative cache, a* **cache state** *is a tuple* (c, h) *where* c *is the capacity (or number of blocks) belonging to the target process, while* h *is the most-recently-used stack position of holes in the LRU stack.*

The range of values for c is $c = 0 \ldots A$ while for h is $h = 1 \ldots A + 1$, where $h = A+1$ indicates that the holes have been completely shifted out of the cache set. An important invariant is $h \leq c + 1$ because the farthest position holes can start is when all c cache blocks of the target process are lumped together starting at the MRU position. As an example, the state is $(3, 1)$ in Figure 2.1(b), $(3, 2)$ in Figure 2.1(c), and $(3, 3)$ in Figure 2.1(d).

The first step in our approach of modeling context switch misses is that for a given state, we find out what events can occur and what the resulting new states can be. To do that, we distinguish two cases: one case in which the right-most hole is not at the LRU position, versus the other case in which the right-most hole is at the LRU position. The former case occurs when

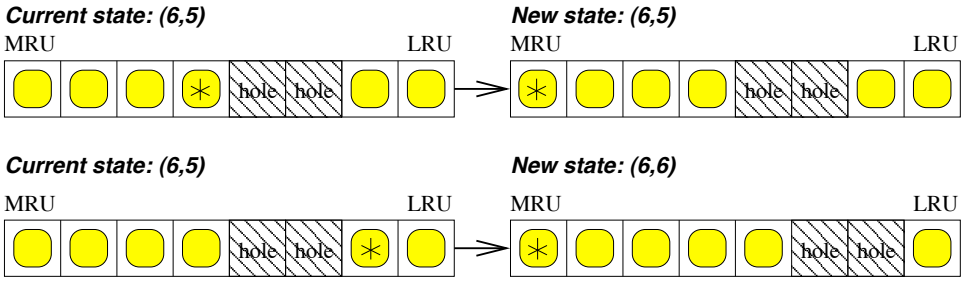


Figure 2.6: The state transition illustration for the case in which the right-most hole is *not* at the LRU position. Accessed blocks are marked by asterisks.

$c \geq h$ and is illustrated in Figure 2.6, while the latter occurs when $h = c+1$ and is illustrated in Figure 2.7. Figure 2.6 shows a process having six cache blocks in an 8-way associative cache set, with two holes starting at the fifth stack position, corresponding to state $(6, 5)$. In the upper case, the process accesses the block to the left of the holes at the fourth stack position (marked by an asterisk), resulting in the block moving to the MRU stack position, but the holes remain where they are. The new state is the same as the current state. In the lower case, the access is to a block to the right of the holes at the seventh stack position, causing the block to move up to the MRU position and shift the holes down one stack position. The new state is now $(6, 6)$.

The second case in which the holes are at the end of the LRU stack is shown in Figure 2.7. The current state is $(6, 7)$. In the upper case, the process accesses the block to the left of the holes at the fourth stack position, resulting in the block moving to the MRU stack position, but the holes remain where they are. The new state is the same as the current state. In the lower case, the access is to a block not found in the LRU stack, causing the block to be brought into the MRU position and the holes shifted down one stack position and the right-most hole shifted out, so the new state is $(7, 8)$.

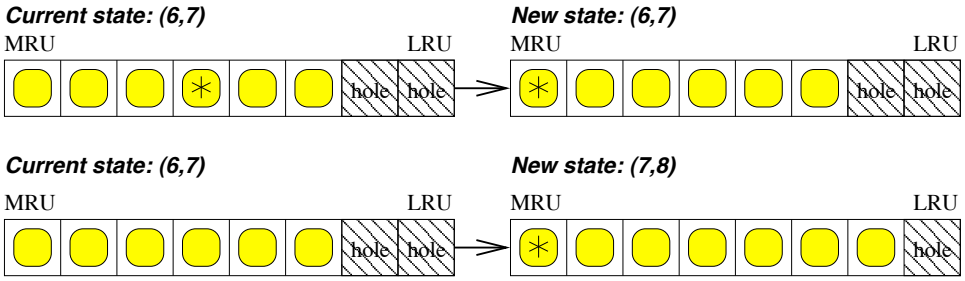


Figure 2.7: The state transition illustration for the case in which the right-most hole is at the LRU position. Accessed blocks are marked by asterisks.

These cases and their state transitions can be captured using Markov modeling. However, rather than starting with a current state and determines what next states it may transition into, we reverse the process and start with a current state and determine what previous states can possibly lead to the current state. In addition, we need to consider one more input variable N, which specifies how many accesses to a cache set a process will make when it resumes execution after a context switch. An iterator variable n will initially take the value of 0 and as the Markov model makes the state transitions on each access, n is incremented until it reaches N. Hence, we modify the definition of a state to include n.

Definition 2 *For a single cache set in an A-way associative cache, a* **cache state** *is a tuple* (c, h, n) *where* c *is the capacity (or number of blocks) belonging to the target process,* h *is the most-recently-used stack position of holes in the LRU stack, and* n *is the number of accesses to the cache set that have occurred so far.*

The Markov model corresponding to the current state (c, h, n) is shown in Figure 2.8. One possibility to reach the current state is when the previous state $(c, h, n-1)$ and the n^{th} cache access uses a block that is to the left of the holes. In this case, the holes *stay* in their previous positions. The probability of such transition is thus the probability of accessing blocks in stack position $1, 2, \ldots, h-1$, i.e. $P_{stay}(h) \equiv \sum_{j=1}^{h-1} P_{reuse}(j)$, where $P_{reuse}(j)$ is obtained from Equation 2.1.

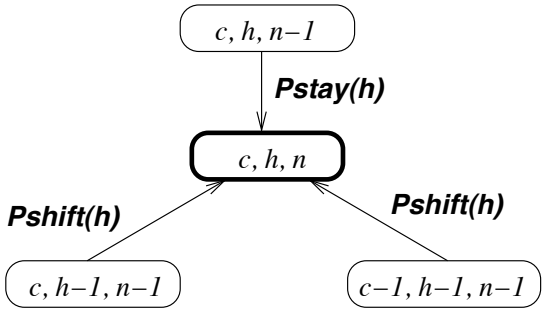


Figure 2.8: The Markov model for what previous states can lead to the current state (c, h, n).

Another possible previous state is $(c-1, h-1, n-1)$. When the rightmost hole is at the LRU stack position (i.e., $h = c+1$), an access that misses in the LRU stack will bring a block into the cache set and *shift* the holes by one position. This case only occurs when the application accesses a block that is not one of the $c-1$ blocks previously in the LRU stack, i.e. the access is to blocks in position $h-1, h, \ldots, \infty$. Hence, the probability of this state transition is $P_{shift}(h) \equiv P_{reuse}(> A) + \sum_{j=h-1}^{A} P_{reuse}(j)$. The final possible previous state is $(c, h-1, n-1)$. When the right-most hole is not at the LRU stack position (i.e., $h < c+1$),

$$P(c,h,n) = \begin{cases} 1 & \textbf{if } c = A-\delta, h = 1, n = 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h) + & \\ P(c-1,h-1,n-1) \times P_{shift}(h) & \textbf{if } c \geq A-\delta, h = c+1, n > 0 \\ P(c,h,n-1) \times P_{stay}(h) + P(c,h-1,n-1) \times P_{shift}(h) & \textbf{if } c \geq A-\delta, h < c+1, n > 0 \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 2.9: Mathematical expression for the probability to reach a state (c, h, n).

an access to a block that is to the right of the holes shifts the holes by one stack position but does not increase the number of blocks of the target process. Since blocks to the right of the holes have stack distances of $h-1, h, \ldots, \infty$, the probability of this state transition is the same as before: $P_{shift}(h) \equiv P_{reuse}(> A) + \sum_{j=h-1}^{A} P_{reuse}(j)$.

Now we would like to express the Markov model in Figure 2.8 with a mathematical expression. Let $P(c, h, n)$ denote the probability of the state (c, h, n) being reached in n accesses. We first note when the process resumes execution, it will find its cache blocks are already shifted by the shift amount δ, where δ is also the number of holes starting from the MRU position. Hence, the initial state is $(A-\delta, 1, 0)$. The probability to be in the initial state is 1, since it is a given state. Hence, the mathematical expression for the Markov model can be written as in Figure 2.9.

The first line in the expression states that the probability to reach the initial state when $n = 0$ is 1. The second line is directly taken from the Markov process in Figure 2.8. When $h = c + 1$, all three previous states can possibly lead to the current state with each own transition probability. The third line is when $h < c+1$. In this case, the previous state cannot be $(c-1, h-1, n-1)$ since the LRU block is not a hole, so there are only two possible previous states: $(c, h-1, n-1)$ or $(c, h, n-1)$. Finally, the fourth line states that the probability value for all other cases is zero. With the expression, $P(c, h, n)$ can be computed recursively for any combination of (c, h, n) values. To determine the number of cache misses in a cache set that a target process likely suffers from, we note that each Markov state has its own probability of a cache miss. If the current state of (c, h, n) is already reached, a new cache access suffers a cache miss when the accessed block is not among the c blocks currently in the LRU stack. Hence, the conditional probability of miss, denoted as $P_{condmiss}(c)$, is the sum of probabilities of accessing blocks in the original stack positions $c+1, c+2, \ldots, \infty$:

$$P_{condmiss}(c) = P_{reuse}(> A) + \sum_{j=c+1}^{A} P_{reuse}(j) \tag{2.2}$$

The contribution of such a miss to the overall cache misses is the product of the probability of reaching the current state and the probability that the current state causes a cache miss in

the next access. Hence,

$$P_{miss}(c,h,n) = P(c,h,n) \times P_{condmiss}(c) \tag{2.3}$$

To get the overall cache misses, all that is left is to sum up the probabilities of cache misses for all combination values of c, h, and n, leading to:

$$TotMisses(N) = \sum_{c=A-\delta}^{A} \sum_{h=1}^{A+1} \sum_{n=0}^{N-1} P_{miss}(c,h,n)$$

The next step in our modeling is to determine which of the total cache misses are context switch misses, and which are natural cache misses. We note that for N accesses, the number of natural cache misses is simply the product of the probability of each cache access incurring a cache miss (absence of holes) times the number of cache accesses, that is:

$$NatMisses(N) = N \times P_{reuse}(> A) \tag{2.4}$$

Therefore, the estimated number context switch misses in a cache set over N accesses is:

$$CSMisses(N) = TotMisses(N) - NatMisses(N) \tag{2.5}$$

Finally, the total estimated context switch misses can be obtained by summing up the estimated context switch misses of all cache sets and over all instances of context switching. For simplicity, in the rest of the paper we will assume that a constant shift amount is applied equally to all cache sets at each instance of context switching.

2.3 Model Validation

2.3.1 Validation Methodology

To validate our model, we compare the number of context switch misses estimated by our model against that obtained from the simulator for SPEC2006 applications. The processor and memory hierarchy models are as described in Section 2.1, except that we use 512KB L2 cache size to reduce the set variation on the global stack distance profile. We chose fourteen applications that have relatively large miss frequency in 50ms time quanta in order to satisfy the assumption that holes are completely shifted out in one time quantum. Hence, we do not include *h264ref, namd* and *povray* in the validation.

We run one application at a time on the simulation model. Every time quantum, we perturb the "real" L2 cache by inserting δ holes to simulate the cache perturbation effect of context switching, while the "isolated" L2 cache is not perturbed. All L1 cache misses are passed to both L2 caches simultaneously. The time quantum is chosen to be 50ms, which corresponds to

a relatively low priority level in Linux OS. After the perturbation, the application is resumed.

Another statistic that is collected over each time quantum is the average number of cache accesses per set, which is used as the input N to our model so that the model can produce an estimate of context switch misses for that time quantum. For validation, the number of context switch misses estimated by the model is compared against that collected by the simulation.

2.3.2 Validation Results

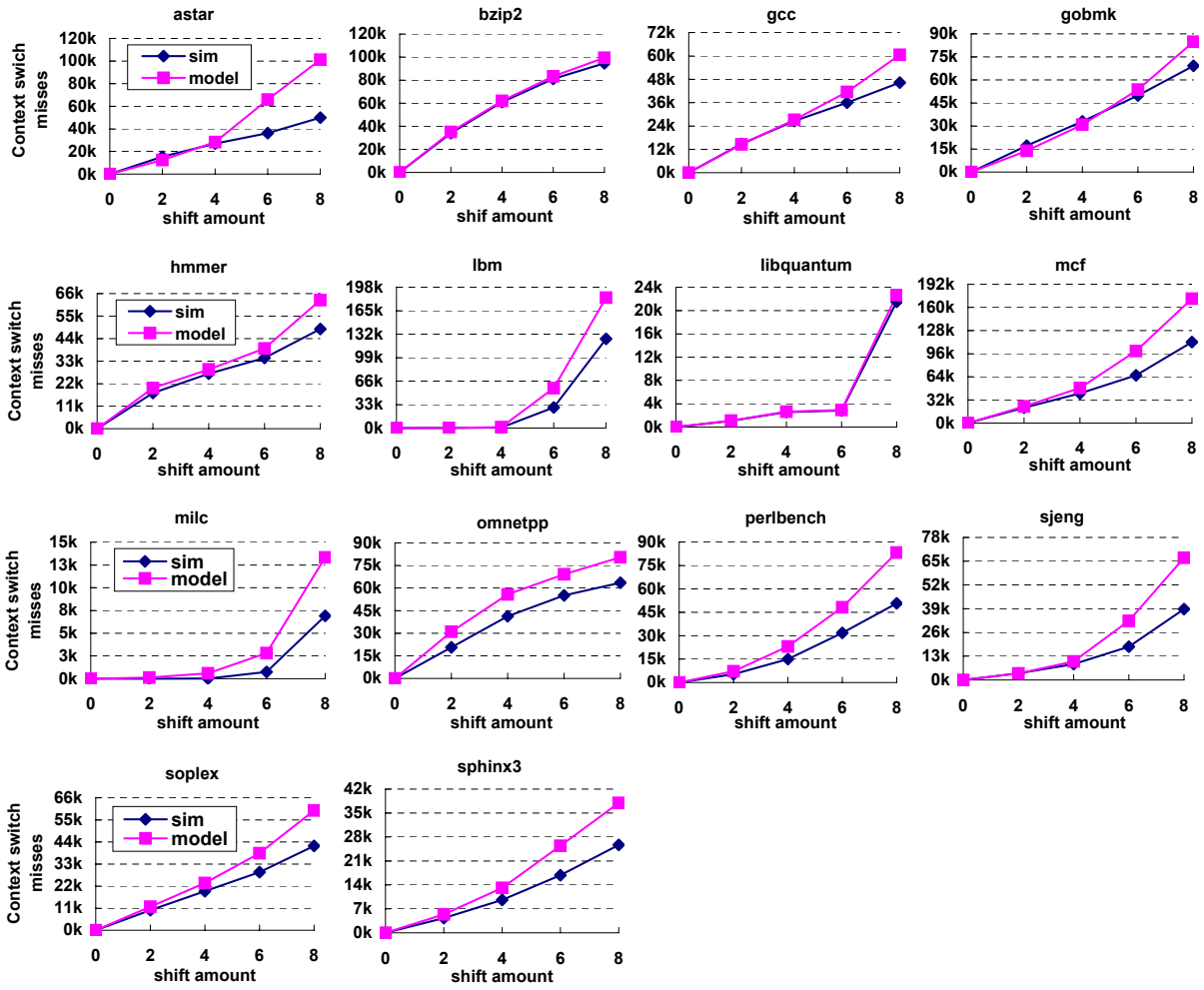


Figure 2.10: Total context switch misses collected by the simulation model versus estimated by our model with variable shift amounts.

Figure 2.10 shows the total number of context switch misses over all instances of context switches as collected by the simulation (labeled as *sim*) and as estimated by our model (labeled

as *model*), with the shift amount (δ) varied from 0 (no cache perturbation – zero context switch misses) to 8 (the entire cache contents are shifted out).

The figure shows that the predicted number of context switch misses presents remarkably similar trends (shapes of the curves) with the actual number of context switch misses. There is some divergence between them when the shift amount is large, but the similarity in the overall trend indicates that our model has captured most of the important variables that affect context switch misses. It also indicates that our model is accurate enough for identifying behavior trends of context switch misses, or for comparing which of two cache configurations yields fewer context switch misses.

We dig deeper into why the predicted number of context switch misses diverges from that collected from simulation when the shift amounts are large. For this purpose, we choose four benchmarks that exhibit the largest estimation errors from Figure 2.11: astar, mcf, milc, and perlbench.

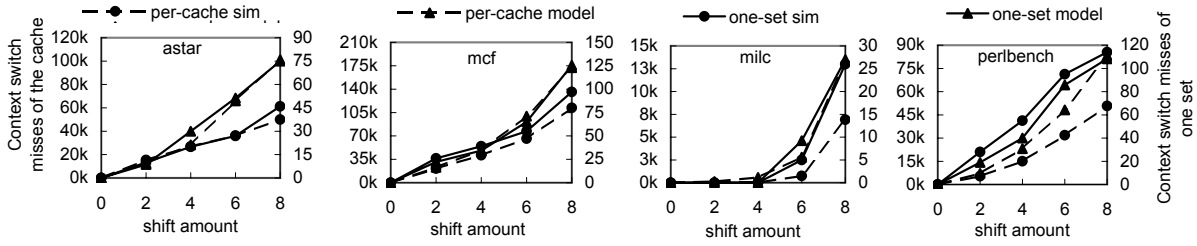


Figure 2.11: Comparing the prediction accuracy of our model using global (entire cache) profile information vs. local (one set) profile information.

First, we suspect that the stack distance profile that we collect for the entire cache may not be representative of the stack distance profile of individual cache sets. It is well-known that a relatively small number of sets have a a disproportionately high number of addresses that map on them (*hot sets*) while the majority of sets have fewer addresses that map on them (*cold sets*). As a result, hot sets have much higher miss rates than cold sets, and the total number of accesses required to shift holes out is much smaller in hot sets than in cold sets. Meanwhile, we use the global stack distance profile, and the number of accesses (i.e. N) fed into the model is the average over all sets. If cold sets outnumber hot sets significantly, N is slightly small for cold sets, but much too large for hot sets. As a result, the number of context switch misses is likely slightly under-estimated for cold sets, but significantly over-estimated for hot sets.

To verify this guess, we choose one set of the cache, collect the local stack distance profile and the number of accesses required to shift all holes in the set out. They are input to the model to predict the number of context switch misses for that set. Figure 2.11 shows predicted

vs. actual number of context switch misses for the entire cache (from Figure 2.10) in dashed lines, super-imposed with the predicted vs. actual number of context switch misses for a single cache set in solid lines. The figure shows that the predicted and actual number of context switch misses are much closer on a single cache set than on the entire cache. This confirms that the error on a large shift amount is mostly caused by the limitation of the input to the model (the global profile information) rather than the inaccuracy of the model itself.

2.3.3 Why Applications Suffer from Context Switch Misses Differently

Figure 2.10 shows that the number of context switch misses increases differently for different benchmarks as the shift amount increases: they may increase slowly at first then rapidly later (such as in lbm), increase at a steady rate (such as in gobmk), or increase rapidly first then slowly later (such as omnetpp). This phenomena is related to why different applications suffer from context switch misses differently (recall the discussion on sjeng vs. hmmer in Section 1.1). The reason for this is the difference in the benchmarks' temporal reuse patterns.

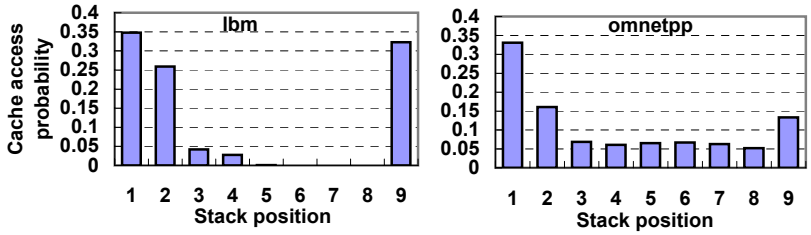


Figure 2.12: Stack distance probability function showing probability of accessing different stack positions.

Figure 2.12 shows the probability function of accesses to different stack positions for lbm and omnetpp. The figure shows that lbm has very low reuse probabilities (almost 0) at the last four stack positions (position 5 through 8), and only reuses cache blocks at the four most recent stack positions (small reuse distance), so a shift amount of four replaces blocks that are not likely reused by lbm, resulting in slow increment of context switch misses. However, a shift amount of eight would replace blocks that are highly reused, so the context switch misses increase rapidly. In contrast, omnetpp highly reuses cache blocks at all stack distances (large reuse distance), so even a small shift amount increases the context switch misses rapidly. As the shift amount increases from six to eight, the MRU and second MRU blocks are also replaced, so at the first glance it might seem that the context switch misses will still rise rapidly. However, note that as the shift amount grows, more blocks are replaced and fewer blocks are reordered. Hence, the number of reordered misses tapers off and becomes zero when the shift amount

is eight, and this in effect significantly slows the growth of context switch misses on larger shift amounts. Overall, the figure points out that different applications react to different shift amounts differently in terms of how much they suffer from cache perturbation.

2.4 Analytical Model Case Studies

We have built a simple but powerful analytical model to study the behavior of context switch misses. In this section, we will show how the model can be used for study while avoiding complicated and exhaustive empirical simulations. In both case studies, we assume the worst case cache perturbation and use the maximum shift amount (δ = L2 cache associativity) for a 512KB cache. We further assume the time quantum to be large enough for the holes introduced by an instance of cache perturbation to be completely shifted out of cache. This corresponds to the worst case context switch penalty on cache performance when a thread resumes execution after context switching [88]. Such a time quantum varies from 1ms to 30ms (average of 10ms) for a SPEC2006 benchmark on a 512KB L2 cache. As in Section 2.3, we use the time quantum of 50ms for all benchmarks. Therefore, the number of cache accesses N for each benchmark is collected during 50ms in our evaluation.

2.4.1 Prefetching

We apply the model to understand the impact of the use of prefetching on the number of context switch misses a thread suffers from. The reason why we choose prefetching is because it is commonly implemented in current processors, but studies in prefetching usually only consider natural misses (cold, capacity, and conflict) and ignore context switch misses.

For the prefetching algorithm, we implement Jouppi's *stream buffers* [45], a popular prefetching algorithm whose variants are implemented in real systems [24, 32]. Stream buffers detect accesses to block addresses that form a sequential or stride pattern (called a *stream*), and prefetch the next few blocks in the stream in anticipation that the processor will continue accessing blocks from that stream. Stream buffers tend to have high *coverage* (a large fraction of cache misses can be prefetched) and high *accuracy* (most prefetched blocks are actually used by the processor), and is relatively simple to implement. In our implementation, the prefetched blocks are placed directly in the L2 cache. For a single stream, up to four blocks can be prefetched. We collect stack distance profiles of each benchmark when it runs on the processor without prefetching and with prefetching respectively, and feed them into the model with other necessary parameters to obtain the predicted results.

Figure 2.13 shows the fractions of the original (natural or total) L2 cache misses that remain across different benchmarks after prefetching is applied, sorted in descending order based on the fraction of natural misses remaining. In other words, applications for which prefetching is

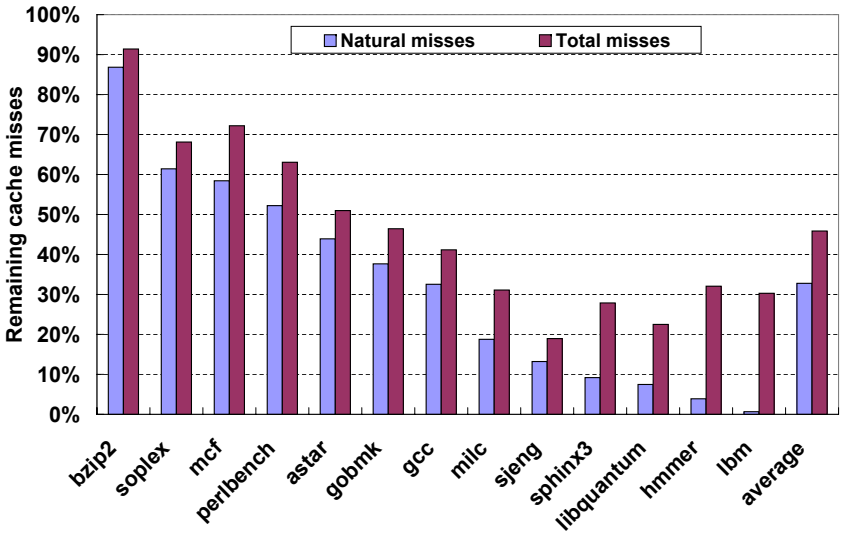


Figure 2.13: The fractions of the original natural and total L2 cache misses that remain after prefetching is applied.

more effective in eliminating natural cache misses are placed on the right. The original total misses are obtained by adding the number of natural misses with the predicted context switch misses, based on the original stack distance profile of each application *without prefetching*. The remaining total misses are obtained by adding the number of natural misses with the predicted context switch misses, based on the stack distance profile of each application *when prefetching is applied*. Note that the divergence error we discuss in Section 2.3 plays little role here since it affects the total misses in both the original and prefetching cases. Also note that the model lumps together all additional fetches as context switch misses, although some of them may be due to additional prefetch requests rather than additional demand misses. We do not need to distinguish them if the focus is the total off-chip bandwidth utilization.

As expected, the figure shows that stream buffer prefetching is highly effective. It removes two thirds of the natural cache misses on average. However, surprisingly in all benchmarks the fraction of remaining total cache misses is consistently higher than the fraction of remaining natural misses (46% vs. 33% on average). The effectiveness of prefetching can sometimes be largely over-stated if context switch misses are not included. In addition, the difference between the fractions of the remaining natural misses versus the remaining total misses is the highest for benchmarks on the right side, i.e. the benchmarks for which prefetching is highly effective in eliminating the natural cache misses.

To understand the reasons behind these observations, recall that prefetching brings blocks into the cache early, causing them to wait in the cache until they are used by the processor. However, while waiting, they gradually shift closer to the LRU position whenever new blocks are

brought into the cache. Consequently, some prefetched blocks are replaced before the processor has a chance to use them. When a processor starts with an "empty" cache (no cache blocks belong to itself) after cache perturbation, it will start demand-fetching and prefetching blocks. All demand-fetched blocks are always useful but some of the prefetched blocks may be replaced before their first use, and have to be refetched later on. In other words, demand-fetched blocks are fetched only once (before they are used), but prefetched blocks may be fetched multiple times (until they are used). Such refetches waste bandwidth and are counted by our model as context switch misses.

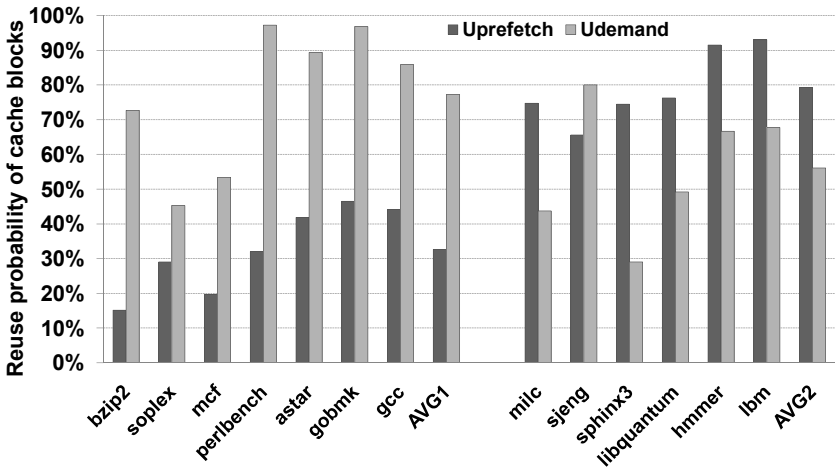


Figure 2.14: Usefulness of a prefetch block: the probability of prefetched blocks to be used ($U_{prefetch}$) vs. usefulness of a demand fetched block: the probability of demand-fetched blocks to be reused (U_{demand}).

For benchmarks that have a large difference between the fractions of the remaining natural misses versus the remaining total misses (milc, sjeng, sphinx3, libquantum, hmmer, and lbm), there is an additional explanation. The reason has to do with how "useful" on average a prefetched block is relative to a demand-fetched block, where "useful" refers to the probability of the processor accessing the block in the near future. To measure such usefulness, let us denote the probability of prefetched blocks to be used by the processor while they reside in the cache as $U_{prefetch}$. In other words, $U_{prefetch}$ is prefetching accuracy. Let us also denote the probability of demand-fetched blocks to be reused by the processor while they reside in the cache as U_{demand}. We measure U_{demand} as the number of cache hits on demand-fetched blocks divided by the total number of cache accesses to demand-fetched blocks. We collect both $U_{prefetch}$ and U_{demand} for

each benchmark and show then in Figure 2.14. The benchmarks are sorted in the same way as in Figure 2.13 but are divided into two groups: the left group consists of benchmarks in which prefetching is not highly effective, while the right group consists of benchmarks in which prefetching is highly effective in eliminating natural cache misses.

Because of the high prefetching effectiveness, the benchmarks on the right group show $U_{prefetch}$ values of at least 65%, and 77% on average. In addition, in most cases, $U_{prefetch}$ exceeds U_{demand} by a significant margin (77% vs. 56% on average). For these benchmarks, a large number of useful prefetched blocks are lost during cache perturbation, requiring them to be refetched again, manifesting in future context switch misses. In contrast, prefetching is not highly effective for the benchmarks on the left group, with U_{demand} having a value of at most 47%, and 33% on average. In addition, $U_{prefetch}$ values for benchmarks on the left are much smaller than U_{demand} values (33% vs. 77% on average). Therefore, for benchmarks on the left group, perturbing prefetched blocks do not result in future context switch misses due to extra prefetching requests. Thus, *perversely*, the more effective the prefetching is, the more context switch misses (or refetches) are incurred. To our best knowledge, the insights that prefetching aggravates context switch misses and that the more effective prefetching is the more context switch misses it incurs, have not been reported elsewhere in literature.

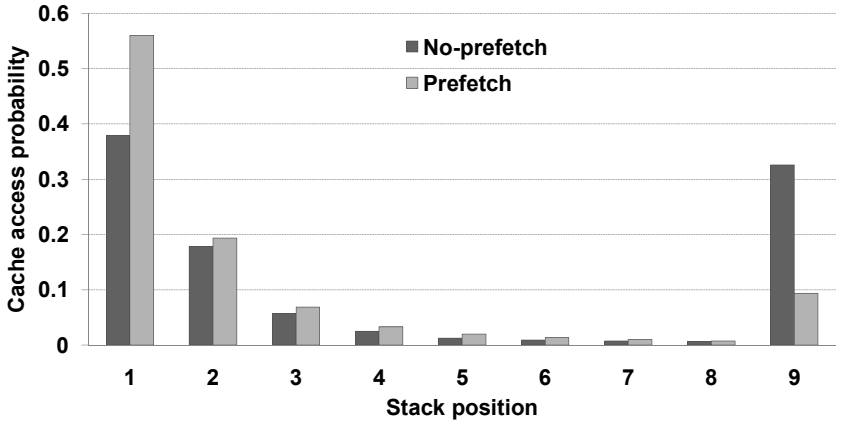


Figure 2.15: Average stack distance profile of SPEC2006 benchmarks without prefetching vs. with prefetching applied.

From the above analysis, we can conclude that prefetching tends to aggravate context switch misses, especially when prefetching is highly effective in reducing the number of natural cache misses. On the other hand, In Section 2.2, we have described that besides the amount of cache

perturbation an application encountered during a context switch, the number of context switch misses it suffers from is solely a function of the temporal reuse pattern of the application. How can we reconcile these two views? The logical hypothesis from reconciling these two views is that prefetching must have altered the temporal reuse pattern of an application in such a way that it increases its susceptibility to context switch misses. As we have discussed in Section 2.3.3, a stack distance profile that has larger reuse distances produces higher number of context switch misses. Does prefetching have such kind of effect on stack distance profiles? We collect the stack distance profile of all benchmarks and find that indeed, prefetching tends to increase reuse probabilities of the stack positions that are affected by cache perturbation. Figure 2.15 shows the average stack distance profile among all benchmarks when there is no prefetching versus when prefetching is applied. As shown in the figure, prefetching increases the reuse distance of cache blocks, especially on medium stack positions (position 2 through 7), therefore it increases the number of context switch misses.

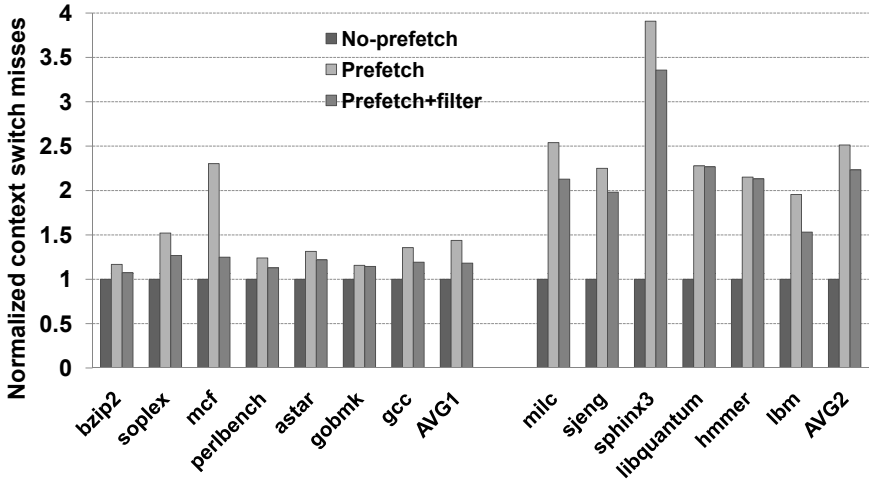


Figure 2.16: Normalized context switch misses without prefetching vs. with prefetching vs. with prefetching plus a filter.

Since now we know that prefetching aggravates the number of context switch misses, a natural question to ask is whether we can reduce prefetching-induced context switch misses through some sort of cache affinity control [39, 40, 43], making the prefetching less aggressive. To test this hypothesis, we add Palacharla's filter [69] to the stream buffer prefetcher to avoid useless prefetches. When there is a cache miss on block address i, instead of beginning prefetching immediately on the next block addresses $i + 1$, $i + 2$, $\cdots$, prefetching is delayed until the

stream is confirmed when we observe a miss on block address $i+1$. To achieve that, a history buffer (filter) with ten entries is used to store the address $i+1$ when a miss on block i occurs. Prefetching starts only when the miss on block $i+1$ is detected. A filter entry is allocated when a miss address does not match in the history buffer and is freed once the stream is detected.

We collect the stack distance profile of each benchmark after applying the previous described filter mechanism to prefetching, and feed the stack distance profiles with a maximum shift amount and number of accesses in 50ms into the model to obtain the predicted number of context switch misses each benchmark suffers from. We show the normalized number of context switch misses for each benchmark in three configurations: without prefetching vs. with prefetching vs. with prefetching plus a filter in Figure 2.16. The figure shows that a less aggressive prefetching (using the filter) generates fewer context switch misses than a more aggressive prefetching. The average increment in the number of context switch misses drops from 44% to 18% for applications on the left group, and from 151% to 123% for applications on the right group.

2.4.2 Cache Sizes

Another case study we try is to investigate how cache size affects the number of context switch misses. The popular perception of enlarging on-chip caches is that it always reduces the number of cache misses. However, such perception is only true when referring to natural cache misses. When we consider the total number of cache misses, a different observation emerges.

More specifically, Figure 2.17 shows the natural and context switch misses for various cache sizes, normalized to the 512KB L2 cache size case for each benchmark. As before, we assume a maximum shift amount and set the time quantum to be 50ms for all cache sizes in the study.

As expected, the figure shows that the number of natural cache misses decreases markedly for most benchmarks as we go from 512KB to 4MB, except for a few benchmarks whose working sets are too big even for a 4MB cache (e.g., lbm, libquantum, and milc). However, the number of context switch misses increases significantly as we go from 512KB to 4MB, because cache perturbation can replace more data in a larger cache, [1] hence it will take more context switch misses in order to refetch some of such data. This observation agrees with previous studies by Agarwal et al. [1], Koka and Lipasti [48], as well as Hwu and Conte [88]. However, our study shows an additional insight in that the growth in the worst-case number of context switch misses can outpace the decline in the number of natural misses for many benchmarks on relatively larger cache sizes. As a consequence, the total number of misses may reach the minimum not at the largest cache size, but at a medium cache size as shown in Figure 2.17.

[1] Although on large cache sizes (1MB to 4MB), time quantum of 50ms may not be large enough to completely shift all holes out, we assume the potential case that more threads can time-share a large cache in a round-robin matter in the system. Therefore, a thread still suffers from the maximum cache perturbation when it resumes execution. We refer this case as the worst-case context switch misses.

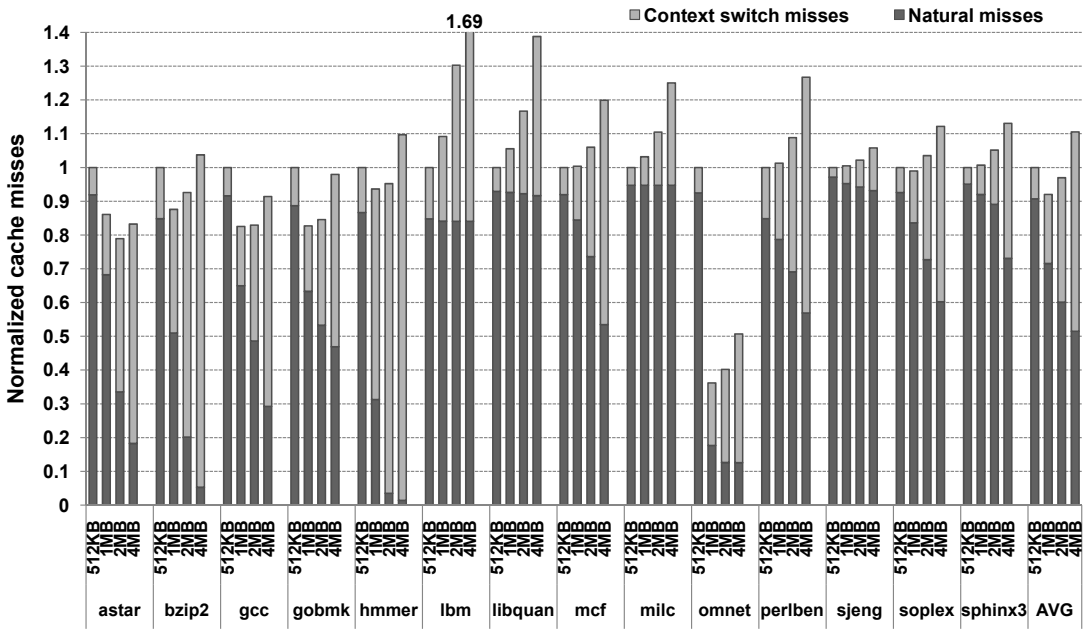


Figure 2.17: Natural and context switch cache misses for various cache sizes with time quantum of 50ms, normalized to the total number of misses on a 512KB L2 cache for each benchmark.

It has been long observed that increasing the cache size has a diminishing return on reducing the number of natural cache misses. For example, in several studies [27, 74], it was shown that for an average commercial workload, the cache size must be quadrupled in order to reduce the cache miss rate by half. In contrast, given a specific time quantum (eg. 50ms) in the system, the worst-case number of context switch misses grows almost linearly with the cache size, as the number of blocks that can be perturbed increases linearly. Therefore, it is inevitable that as the cache size increases (while keeping the time quantum fixed), at some point, the worst-case number of context switch misses will eclipse the number of natural misses.

Recall that in Section 2.1.3 we made an observation that when the time slice of the primary benchmark is increased to $r\times$, the number of context switch misses it suffers from decreases to $\frac{1}{r}$ of the original number of context switch misses on average (as shown in figure 2.3). Note that the L2 cache size in the experiment was fixed to 1MB. If the cache size increases at the same pace with the increase in the time quantum, will the context switch misses still show the same trend? The answer is no. Figure 2.18 shows the normalized natural and context switch cache misses for various cache sizes and time quanta. Specifically, the time quantum is increased to twice as much when the cache size is doubled. The results are averaged among all SPEC2006 benchmarks. In Figure 2.18(a), the natural misses decrease almost linearly with the

increase in cache size; however, the number of context switch misses decreases at a much slower pace. From 512KB cache with 5ms time quantum to 1MB cache with 10ms time quantum, the number of context switch misses does not decline at all. The reason is because the benefit of reduction in context switch frequency is offsetted by the increment in the number of context switch misses for a single context switch instance. Consequently, when we look at the fraction of context switch misses with regard to the total number of cache misses for each configuration in Figure 2.18(b), the fraction increases rapidly first (from 512KB, 5ms to 1MB, 10ms) and then tends to saturate (up to 57%) at very large cache size and time quantum.

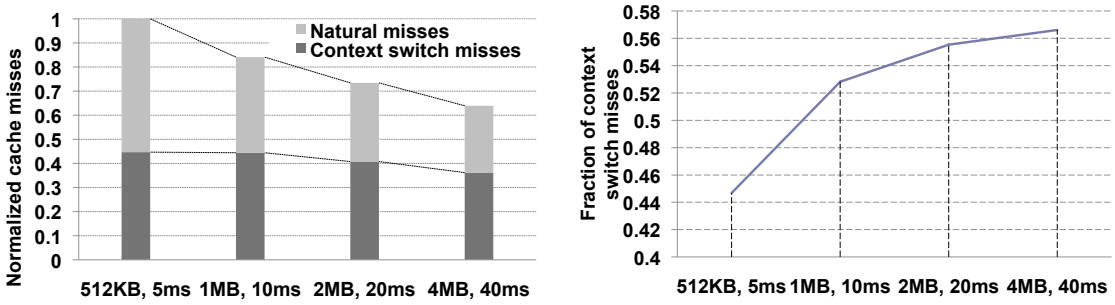


Figure 2.18: Average natural and context switch cache misses of SPEC2006 benchmarks for various cache sizes and time quanta. (a) shows the split misses normalized to a 512KB L2 cache with 5ms case; (b) shows the fraction of context switch misses with regard to the total cache misses for each configuration.

Clearly, this result points out that careful design considerations must be employed in order to keep the actual number of context switch misses much below the worst-case number of context switch misses. The observation above has several significant implications for OS thread scheduler design. First, the choice of a time slice to use for a given application must not only consider the application thread's priority, but also the size of the cache in the system. While using a large time slice works well to minimize the number of context switch misses on a relatively small cache size through reducing the frequency of context switches, it may increase the number of context switch misses for larger cache sizes due to larger amount of cache perturbation. The worst possible choice of a time slice is equal to the time interval needed for an application to fill the entire cache. At such a time slice, the cache is consistently in the transient state of being perturbed or refilled. Consequently, our results argue that for a large cache size, the time slice must be chosen to be relatively small (so that the cache is only partially perturbed by a context switch) or very large (so that the cache can stay filled up and reuse blocks for a while before being perturbed).

2.5 Related Work

There have been many empirical studies aiming at understanding the magnitude and behavior of context switching overheads [16, 22, 48, 49, 52, 63, 87]. Mogul and Borg [63] evaluated the impact of context switching on the cycles per instruction (CPI) and cache miss rates for various cache parameters and found that the overheads for an average context switch could be up to tens to hundreds of microseconds. Kwak et al. [49] evaluated multithreaded systems and showed that fast context switching could improve performance by exploiting data locality. Koka and Lipasti [48] characterized context switch misses of an application for various cache parameters and investigated how a potentially optimum scheduling policy could reduce them. Li et al. [52], and Fromm and Trehaft [22] measured both direct and indirect context switching overheads through simulation and concluded that indirect context switch overheads due to the cache perturbation effect are much more significant than direct overheads. Tsafrir [87] and David et al. [16] performed similar experiments on Intel Pentium and ARM platforms respectively and arrived at the same conclusion. Li et al. [52] also showed that the working set and data access patterns of an application could significantly affect the context switch overheads.

Our study complements the findings from prior studies. While prior studies did not identify context switch misses as consisting of two types (replaced and reorder misses), our study identifies reordered misses as context switch misses and establishes the interaction between them and the LRU cache replacement policy, cache size, and the amount of cache perturbation.

In addition, previously there were no studies that could show the exact relationship between an application's temporal reuse pattern with its vulnerability from suffering context switch misses. Empirical studies, either simulation studies or measurement on real systems, cannot reveal the underlying mathematical relationships between the various factors that affect the context switch misses and the number of context switch misses an application suffers from.

Analytical models can potentially fill that role but they need to capture all essential variables in order to have a sufficient resolution. Several analytical models have been proposed in the past [1, 23, 83, 86, 88]. Thiebaut and Stone [86] modeled context switch misses by considering the cache size and the working set size of an application in a time-shared processor environment. However, their model ignores the impact of the temporal reuse patterns of the application.

Agarwal et al. [1] proposed an analytical model to predict the overall cache miss rate including context switch misses. Agarwal's model takes into account the temporal reuse patterns of applications, but it ignores the interaction of cache replacement policy and the behavior of context switch misses. The model only captures the replaced context switch misses and ignores the reordered context switch misses.

Hwu and Conte [88] proposed a high-level model to predict the worst-case number of context switch misses given benchmark traces. The worst-case number is obtained by assuming that

cache perturbation during a context switch replaces all existing data of an application in the cache. In contrast, our model can predict the number of context switch misses under various degrees of cache perturbation. Hwu and Conte also proposed a method to estimate the degree of cache perturbation by estimating the fraction of cache content flushed during a context switch for a given benchmark. The degree of cache perturbation can be a useful input to our model.

Suh et al. [23, 83] proposed an analytical model for estimating the total miss rate of an application that includes all types of context switch misses. Unfortunately, the model requires a *continuous miss rate curve* as profiling information, i.e. the miss rate of a thread for any given cache size, including non-integer values. Such profiling information is difficult to obtain since cache parameters (size, associativity, block size) have discrete rather than continuous values. While discrete data series can be interpolated into continuous one, the interpolation only produces accurate modeling if there are many data points. Hence, the model in [23, 83] assumes a small cache that is fully-associative, which is an unrealistic assumption for L2 or L3 caches, in which the overheads due to context switch misses are more significant than in a L1 cache.

The model proposed in the second part of our study models the relationship between temporal reuse patterns of applications, cache configurations, and the LRU cache replacement policy. It includes all types of context switch misses, relies on a realistic profiling information, and is appropriate for modeling large and set associative caches.

2.6 Conclusions

We have characterized the behavior of context switch misses, especially focusing on the behavior of reordered misses. We have shown that reordered misses occur due to the interaction of cache replacement policy with the temporal reuse pattern of an application. The number of reordered misses tends to reach its peak when the cache perturbation affects roughly a half of the cache size, and the fraction of reordered misses of the total context switch misses increases as the cache size increases.

We have also presented an analytical model that reveals the mathematical relationship between cache parameters, the temporal reuse behavior of a thread, and the number of context switch misses the thread suffers from. We found that applications with a flatter stack distance profile are more vulnerable from suffering context switch misses than applications with a concentrated stack distance profile. Applying the model, we showed that prefetching, because it makes stack distance profile flatter, aggravates context switch misses. In addition, perversely, the more effective the prefetching is for an application, the higher the number of context switch misses the application suffers from. We found that a less aggressive prefetcher can alleviate the context switch misses. We also showed that the worst case number of context switch misses

an application suffers from tends to increase proportionally with cache sizes, to the extent that may completely negate the reduction in other types of cache misses.

Many computer architecture studies focus solely on cold, capacity, and conflict misses. However, our findings suggest context switch misses are an important part of the picture and should be included in the studies of cache designs. Our model can serve as a useful tool for analyzing the behavior of context switch misses for various applications.

Chapter 3

Effects of Bandwidth Partitioning on CMPs Performance

This chapter is organized as follows: Section 3.1 describes the construction of the model and shows how optimum bandwidth partitioning is derived, Section 3.2 discusses exploration using the model to understand important factors affecting performance improvement due to bandwidth partitioning and the interaction between cache partitioning and bandwidth partitioning, Section 3.3 validates the findings through a simulation-based empirical evaluation, Section 3.4 reviews the related work, and Section 3.5 summarizes the findings in this chapter [57].

3.1 Analytical Bandwidth Partitioning Model Formulation

3.1.1 Assumptions

For this study, we assume a base CMP system configuration with several cores, in which each core has a private L1 instruction and data cache (Figure 3.1(a)). The L2 cache and off-chip bandwidth are shared by all cores (although our model is applicable for private L2 caches as well). Off-chip memory requests are served by the off-chip interface in a First Come First Served (FCFS) manner through a single queue. The L2 cache can be partitioned between threads that run on different cores. We assume the cores run threads from independent and sequential applications that do not share data with one another.

By *bandwidth partitioning*, we specifically refer to allocating shares (fractions) of off-chip bandwidth between different cores, without changing the memory controller scheduling policy. Note that naively allocating fixed fractions of off-chip bandwidth to different cores is risky because it can degrade, instead of improve, system performance, for two reasons. First, if a thread is allocated more bandwidth than it needs, the allocated bandwidth will be underutilized, while at the same time other threads may be penalized due to insufficient bandwidth allocation. Secondly, bandwidth usage is often bursty; hence a fixed partition can significantly hurt performance during bursts of memory requests, and suffer from fragmentation during a

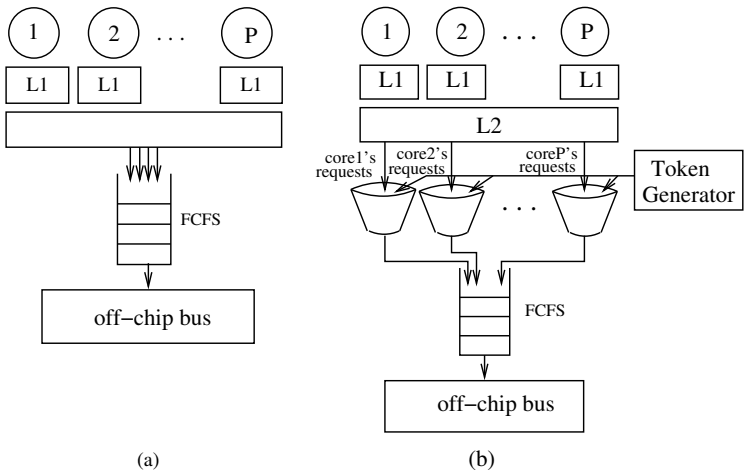


Figure 3.1: The assumed base CMP configuration (a), and configuration with token bucket bandwidth partitioning (b).

period of sparse memory requests. To allocate fractions of off-chip bandwidth to different cores while tolerating the bursty nature of memory requests, we propose and assume that bandwidth partitioning is implemented using a *token bucket algorithm* (Figure 3.1(b)), a bandwidth partitioning algorithm borrowed from computer networking [85].

With the token bucket algorithm, each off-chip memory request is allowed to go to the off-chip interface only when it has a matching token in the respective bucket. There is a token generator that distributes tokens to different buckets at the rates proportional to the fractions allocated to different cores. Unused tokens are accumulated in the buckets and are consumed in the future during a period of bursty memory requests. Thus, the depth of the buckets determines the extent of how much burstiness can be tolerated. When tokens for a core are discarded because the bucket is full, they are lost forever; hence the core will never reach the allocated bandwidth fraction. In order to avoid mismatches between the fraction of allocated bandwidth and the fraction of actual usage, for modeling purposes, we assume an unlimited token bucket size for each core. In Section 3.3.2, we will explain how the algorithm can be practically implemented for memory bandwidth partitioning using limited token bucket sizes.

We assume that the goal of performing off-chip bandwidth partitioning is to optimize the overall system performance expressed in weighted speedup. Weighted speedup was proposed by Snavely and Tullsen [3] to measure the speedup of a co-schedule consisting of multiple threads running together, compared to the case in which each thread runs alone in the system. More precisely, if thread i achieves an Instruction Per Cycle (IPC) of $IPC_{alone,i}$ when running alone in a CMP system, and achieves an IPC of IPC_i when running in a co-schedule in which other applications run simultaneously in other cores, the weighted speedup (WS) is the sum of

individual IPC speedups (or sum of individual CPI slowdowns):

$$WS = \sum_{i=1}^{N} \frac{IPC_i}{IPC_{alone,i}} = \sum_{i=1}^{N} \frac{CPI_{alone,i}}{CPI_i} \tag{3.1}$$

Although the overall system performance can be expressed in different metrics, we choose weighted speedup because it is a very widely used metric in studying system performance of co-scheduled applications, and includes some measure of fairness, in which speeding up one application at the expense of others will have some offsetting effect. We realize that other aspects (power, energy, fairness, QoS) are important as well; however they are beyond the scope of this paper.

In our model, we take into account off-chip memory requests due to L2 cache misses and ignore other requests such as write backs, prefetches, and coherence requests. Besides being much fewer and less frequent than L2 cache misses, write backs are not in the critical path of performance, hence they have no direct impact on weighted speedup. Off-chip coherence traffic is ignored since we only consider applications that do not share data, running on a single CMP. Prefetching is ignored to keep the model simple, but we will show that the model can easily be adapted to include prefetching requests in Chapter 4. Finally, write backs, prefetches, and coherence requests compete with regular cache misses in accessing the off-chip bandwidth. Thus, their inclusion would make off-chip bandwidth scarcer, and would increase the need for bandwidth partitioning.

3.1.2 Model Formulation

The input to our analytical model includes system parameters as well as per-thread parameters, listed in Table 3.1.

Table 3.1: Input and parameters used in our model.

System parameters	
f	CPU clock frequency (Hz)
K	Cache block size (Bytes)
B	Peak off-chip memory bandwidth (Bytes/sec)
N	Number of cores in the CMP
Thread-specific parameters	
M_i	Thread i's L2 cache miss rate
A_i	Thread i's L2 cache access frequency (#accesses per second)
$CPI_{L2\infty,i}$	Thread i's CPI assuming infinite L2 cache size
$CPI_{alone,i}$	Thread i's CPI when it runs alone in the CMP
$t_{m,i}$	Thread i's average L2 miss penalty (#cycles)
β_i	Fraction of off-chip bandwidth assigned to thread i

Our model is based on the additive CPI model used in modeling cache performance of uniprocessor systems [20, 59, 78]. We extend the model to a CMP system in which multiple threads run together and result in contention for the off-chip bandwidth. We model the off-chip bus as a queuing system [55], in which contention results in queuing delay for off-chip memory requests.

Extending Uniprocessor CPI Model to a CMP System

The CPI model [20, 59, 78] states the average number of cycles it takes to execute an instruction in a uniprocessor with two levels of caches can be expressed as the addition of CPI assuming an infinite L2 cache and the extra CPI due to L2 cache misses:

$$CPI = CPI_{L2\infty} + h_m \cdot t_m \tag{3.2}$$

where $CPI_{L2\infty}$ is the CPI of the program assuming L2 cache size is infinite in the system, h_m is the average number of L2 cache misses per instruction, and t_m is the average L2 miss penalty. Note that t_m is not necessarily equal to the latency to access the off-chip memory since it implicitly includes the effect of instruction-level parallelism (ILP) and memory-level parallelism (MLP). For example, for a superscalar out-of-order processor, t_m will be smaller than for an in-order processor, because a higher degree of ILP overlaps miss time with execution of independent instructions, and a higher degree of MLP amortizes the miss time over multiple misses. In addition, the use of prefetching may decrease both h_m as well as t_m given the available memory bandwidth is plentiful enough in the system.

The off-chip interface can be thought of as a single-server queuing system which serves requests on a first come first serve basis (FCFS). Requests that find the bus busy must wait in a queue until the bus is free. Thus, the effect of off-chip bandwidth contention is the extra queuing delay suffered by off-chip memory requests. Suppose that the extra queuing delay due to contention between multiple cores is $\Delta t_{m,i}$ for an application thread i, (the queuing delay due to contention between requests from a single core is already reflected in $t_{m,i}$). The CPI model for thread i can be expressed as:

$$CPI_i = CPI_{L2\infty,i} + h_{m,i} \cdot (\Delta t_{m,i} + t_{m,i}) \tag{3.3}$$

From the equation, we can see that the effect of the extra queuing delay ($\Delta t_{m,i}$) on CPI_i depends on $h_{m,i}$. Since the value of $h_{m,i}$ is different for different applications, different applications suffer from the queuing delay to different degrees. Hence, bandwidth partitioning can improve the overall system performance if it favors applications that are more sensitive to queuing delay (i.e., having a large $h_{m,i}$ value) over applications that are less sensitive to queuing delay.

Note that h_m is the number of L2 misses per instruction. It is not an independent variable as it is affected by CPI, L2 miss rate (M), L2 access frequency (A) and CPU frequency (f). More specifically, h_m can be expressed as the multiplication of miss frequency ($M \cdot A$) and average time taken to execute one instruction ($\frac{CPI}{f}$):

$$h_m = \frac{M \cdot A \cdot CPI}{f} \tag{3.4}$$

Since f is a system parameter, only M, A, and CPI are thread-specific parameters. Substituting h_m from Equation 3.4 into the CPI in Equation 3.3, and solving for CPI, we can express CPI of thread i as a function of its miss frequency ($M_i \cdot A_i$):

$$CPI_i = \frac{CPI_{L2\infty,i}}{1 - \frac{M_i \cdot A_i}{f} \cdot (\Delta t_{m,i} + t_{m,i})} \tag{3.5}$$

which leads us to the next observation:

Observation 1 *The sensitivity of an application's performance to queuing delay for their off-chip memory requests depends on three variables: its miss frequency ($M_i \cdot A_i$), its CPI assuming an infinite L2 cache size ($CPI_{L2\infty,i}$), and the average penalty for off-chip memory requests ($t_{m,i}$).*

In the next step, we will attempt to derive the value of $\Delta t_{m,i}$ using Little's law [55] for two cases: a system without bandwidth partitioning, and a system with bandwidth partitioning. Then, we will compare them to obtain more insights.

System without Bandwidth Partitioning

Let λ denote the arrival rate of cache misses from all cores at the off-chip interface queue. Since all cores generate cache misses, λ is the sum of L2 miss frequencies ($M_i A_i$) from all threads that run simultaneously on different cores:

$$\lambda = \sum_{i=1}^{N} M_i A_i \tag{3.6}$$

Little's law [55] for a queuing system states that the average queue length (N) is equal to the arrival rate (λ) multiplied by the system time (T), i.e. $N = \lambda \cdot T$. If we denote the total available bandwidth as B Bytes/sec and cache block size as K bytes, the service time of a cache miss request on the off-chip bus is $\frac{K}{B}$ seconds. Replacing T in Little's law with K/B and λ with Equation 3.6, we can express the average number of memory requests (N) arriving during the service time as:

$$N = \frac{K}{B} \sum_{i=1}^{N} M_i A_i \tag{3.7}$$

Since the average waiting time of a newly arriving memory request (denoted as W) in the system is the total service time of N requests that are ahead of it in the queue, and using N from Equation 3.7, W (in cycles) can be computed as:

$$W = f \cdot \frac{K}{B} \cdot N = f \cdot \frac{K^2}{B^2} \cdot \sum_{i=1}^{N} M_i A_i \tag{3.8}$$

Note that CPI_i is calculated from the thread i's perspective. For a specific thread i, the waiting time must be adjusted by the probability that the request is from thread i, because the thread has to wait longer for N of its own requests. Since requests arrive according to the arrival rates of different threads, the probability of a request is from thread i is the share of the arrival rate of i from the total arrival rates, i.e. $\frac{M_i A_i}{\sum_{j=1}^{N} M_j A_j}$. Thus, the expected waiting time for requests from thread i due to contention from multiple cores is:

$$\Delta t_{m,i} = \frac{W}{M_i A_i / \sum_{j=1}^{N} M_j A_j} = \frac{\left(\sum_{j=1}^{N} M_j A_j\right)^2 \cdot f \cdot K^2}{M_i \cdot A_i \cdot B^2} \tag{3.9}$$

Substituting $\Delta t_{m,i}$ in Equation 3.9 into the CPI expression in Equation 3.5, CPI of thread i can now be expressed as:

$$CPI_i = \frac{CPI_{L2\infty,i}}{1 - \frac{M_i A_i \cdot t_{m,i}}{f} - \frac{\left(\sum_{j=1}^{N} M_j A_j\right)^2 \cdot K^2}{B^2}} \tag{3.10}$$

System with Bandwidth Partitioning

Now we will derive the CPI of a thread assuming the bandwidth is partitioned. Let β_i denote the fraction of bandwidth allocated to thread i. Since the sum of fractions of bandwidth for all threads must be 100%, we require that $\sum_{i=1}^{N} \beta_i = 1$. From the point of view of thread i, the off-chip bandwidth it can use is the fraction of bandwidth allocated to it multiplied by the total peak bandwidth, i.e. $B \cdot \beta_i$ Bytes/sec. Based on this bandwidth, thread i's service time is $T_i = \frac{K}{B\beta_i}$ seconds. The arrival rate of requests from thread i is $\lambda_i = M_i A_i$. Applying Little's law for the effective bandwidth of thread i, the average number of memory requests waiting in the queue and average queuing delay for the request from thread i are:

$$N_i = M_i A_i \cdot \frac{K}{\beta_i B} \tag{3.11}$$

$$\Delta t_{m,i} = N_i \cdot T_i \cdot f = \frac{M_i A_i \cdot K^2 \cdot f}{\beta_i^2 \cdot B^2} \tag{3.12}$$

Substituting Equation 3.12 into Equation 3.5, CPI for thread i can then be expressed as:

$$CPI_i = \frac{CPI_{L2\infty,i}}{1 - \frac{M_i A_i \cdot t_{m,i}}{f} - \frac{(M_i A_i)^2 \cdot K^2}{\beta_i^2 \cdot B^2}} \tag{3.13}$$

Comparing Equation 3.13 and Equation 3.10, we can observe that Equation 3.10 is a special case of Equation 3.13 where:

$$\beta_i = \frac{M_i A_i}{\sum_{j=1}^{N} M_j A_j} \tag{3.14}$$

which leads us to the following observation:

Observation 2 *In a CMP system that has unregulated off-chip memory bandwidth usage between multiple cores, the off-chip memory bandwidth is naturally partitioned between cores, where the natural share of off-chip bandwidth a core uses is equal to the ratio of miss frequency of the core to the sum of all miss frequencies of all cores.*

The observation certainly sounds intuitive and logical. In this case, our model validates the intuition.

Optimum Bandwidth Partitioning

In this section, we will derive the theoretical optimum bandwidth partitioning assuming our goal is to maximize weighted speedup (Section 3.1.1). Substituting CPI from Equation 3.13 into the weighted speedup formula (Equation 3.1), we obtain:

$$WS = \sum_{i=1}^{N} \frac{CPI_{alone,i}}{CPI_{L2\infty,i}} \cdot \left(1 - \frac{M_i \cdot A_i \cdot t_{m,i}}{f} - \frac{(M_i A_i)^2 \cdot K^2}{\beta_i^2 \cdot B^2}\right) \tag{3.15}$$

Given that the first and second terms in the equation are not affected by bandwidth partitioning, to maximize weighted speedup, we need to minimize the sum of the third terms in the above equation, i.e.:

$$f(\beta_1, \beta_2, \ldots, \beta_N) = \sum_{i=1}^{N} \frac{CPI_{alone,i}}{CPI_{L2\infty,i}} \cdot \frac{(M_i A_i)^2 \cdot K^2}{\beta_i^2 \cdot B^2} = \sum_{i=1}^{P} k_i \beta_i^{-2} \tag{3.16}$$

where $k_i = \frac{CPI_{alone,i}}{CPI_{L2\infty,i}} \cdot \frac{(M_i A_i)^2 K^2}{B^2}$.

Minimizing $f(\beta_1, \beta_2, \ldots, \beta_N)$ is a constrained optimization problem, with a goal function of Equation 3.16 and constraint function $\sum_{i=0}^{N} \beta_i = 1$. To solve the problem, we can apply Lagrange multipliers [31] by introducing a new variable (r) and a Lagrange function defined by:

$$L(\beta_1, \beta_2, \ldots, \beta_n, r) = f(\beta_1, \beta_2, \ldots, \beta_n) + r(\sum_{i=1}^{N} \beta_i - 1) \tag{3.17}$$

Differentiating $L(\beta_1, \beta_2, \ldots, \beta_n, \lambda)$ with respect to β_i and λ, we obtain the following partial differential equations:

$$\begin{cases} \frac{\partial L}{\partial \beta_i} = -2 \cdot k_i \cdot \beta_i^{-3} + \lambda = 0 & i = 1, 2, \ldots, n \\ \frac{\partial L}{\partial r} = \beta_1 + \beta_2 + \ldots + \beta_n - 1 = 0 \end{cases}$$

Solving the differential equations, we arrive at bandwidth partition for thread i (β_i) that maximizes weighted speedup:

$$\beta_i = \frac{k_i^{1/3}}{\sum_{j=1}^{P} k_j^{1/3}} = \frac{(M_i A_i)^{2/3} (\frac{CPI_{alone,i}}{CPI_{L2\infty,i}})^{1/3}}{\sum_{j=1}^{N} (M_j A_j)^{2/3} (\frac{CPI_{alone,j}}{CPI_{L2\infty,j}})^{1/3}} \tag{3.18}$$

leading us to the next observation:

Observation 3 *Weighted speedup-optimum bandwidth partition for a thread can be expressed as a function of all co-scheduled threads' miss frequencies, infinite-L2 CPIs, and CPIs when each thread runs alone in the system as shown in Equation 3.18.*

Note that the parameters in Equation 3.18 may vary due to the changes in the behavior of an application throughout its execution. The equation does not assume them to have constant values. However, it assumes that the parameter values for any part of the application are available.

Let us examine Equation 3.18 in more details. Basically, the equation shows that there are two main factors affecting the optimum bandwidth partition: (1) miss frequencies of all threads, and (2) the ratios of CPI for running alone to CPI for infinite-L2. Between the two factors, the miss frequencies are likely the dominant contributor, for several reasons. First, the CPIs contribute as a ratio, hence the CPI bias of an application (very high or very low CPI) is somewhat neutralized when we take a ratio. Second, the CPI ratio is exponentiated by a smaller power ($\frac{1}{3}$) versus a larger power ($\frac{2}{3}$) in miss frequencies. Finally, when there is a large number of cores sharing the last level cache, the size of the cache is usually large enough to make the CPI when a thread runs alone in the system close to the CPI when assuming an infinite cache. As a result, $CPI_{alone,i}$ will approach $CPI_{L2\infty,i}$ and their ratio will approach 1, [1] leading to a simpler equation:

$$\beta_i = \frac{(M_i A_i)^{2/3}}{\sum_{j=1}^{N} (M_j A_j)^{2/3}} \tag{3.19}$$

Table 3.2 compares the natural bandwidth that share thread i uses when off-chip bandwidth usage is unmanaged (Equation 3.14), versus the optimum bandwidth partition for the thread (Equation 3.18) for N number of cores, and when we assume N is very large.

[1] For a SPEC2006 benchmark running alone on a four-core CMP with 2MB L2 cache size, $(\frac{CPI_{alone,i}}{CPI_{L2\infty,i}})^{1/3}$ ranges from 1.000 (h264ref or povray) to 1.573 (mcf), and around 1.133 on average.

Table 3.2: Comparing natural and optimum bandwidth sharing for any N and when N is large.

Case	Natural Share	Optimum Bandwidth Partition	
		No Assumption on the Value of N	Large N
Equal M_iA_i and $\frac{CPI_{L2\infty,i}}{CPI_{alone,i}}$	$\frac{1}{N}$	$\frac{1}{N}$	$\frac{1}{N}$
Equal M_iA_i only	$\frac{1}{N}$	$\frac{(CPI_{alone,i}/CPI_{L2\infty,i})^{1/3}}{\sum_{j=1}^{N}(CPI_{alone,i}/CPI_{L2\infty,i})^{1/3}}$	$\frac{1}{N}$
Equal $\frac{CPI_{L2\infty,i}}{CPI_{alone,i}}$ only	$\frac{M_iA_i}{\sum_{j=1}^{P}M_jA_j}$	$\frac{(M_iA_i)^{2/3}}{\sum_{j=1}^{N}(M_jA_j)^{2/3}}$	$\frac{(M_iA_i)^{2/3}}{\sum_{j=1}^{N}(M_jA_j)^{2/3}}$
Nothing equal	$\frac{M_iA_i}{\sum_{j=1}^{P}M_jA_j}$	$\frac{(M_iA_i)^{2/3}(CPI_{alone,i}/CPI_{L2\infty,i})^{1/3}}{\sum_{j=1}^{N}(M_jA_j)^{2/3}(CPI_{alone,j}/CPI_{L2\infty,j})^{1/3}}$	$\frac{(M_iA_i)^{2/3}}{\sum_{j=1}^{N}(M_jA_j)^{2/3}}$

If all co-scheduled applications have equal miss frequencies, and equal ratios of CPI (CPI when running alone divided by CPI with infinite L2 cache), then the optimum bandwidth partitioning is $\frac{1}{N}$, which is the same as its natural bandwidth share. This implies that unmanaged bandwidth usage already achieves a weighted speedup-optimum partition and hence bandwidth partitioning can not improve weighted speedup further. When N is large, interestingly, only two cases matter. The first is when miss frequencies of all threads are equal, then both natural bandwidth shares and optimum bandwidth partitions are equal. Otherwise, they differ in the exponentiation of the miss frequencies. Natural bandwidth share uses an exponent of 1, whereas the optimum bandwidth partition uses an exponent of $\frac{2}{3}$. We will explore the consequences in the next section.

3.2 Impact of Bandwidth Partitioning on System Performance

In the previous section, we have presented how we construct the analytical model for how bandwidth partitioning affects system performance. In this section, we will use the model to gain further understanding into the nature of system performance impact from bandwidth partitioning.

We assume a two-core CMP with two threads running on them. We also assume that the CPI ratios of both threads are equal to 1 for simplicity of discussion, i.e. $\frac{CPI_{alone,1}}{CPI_{L2\infty,1}} = \frac{CPI_{alone,2}}{CPI_{L2\infty,2}} = 1$. However, we note that the trends shown in the figures in this section are the same regardless of the simplifying assumption. Hence, the validity of the observations does not depend much on the use of the assumption.

Since the optimum bandwidth is now a function of miss frequencies of both threads, Figure 3.2 plots the optimum bandwidth vs. natural bandwidth share for thread 2 as a function of the ratio of miss frequencies of thread 2 to thread 1. From the figure, we could see that when the miss frequency ratio is 1, the natural bandwidth share for thread 2 stands at 50%, equal to the optimum bandwidth partition. Hence, when threads have equal miss frequencies, their natural bandwidth shares already achieve an optimum weighted speedup. As the miss frequencies

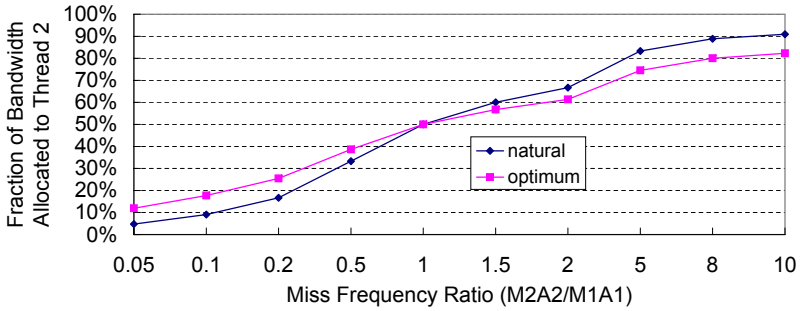


Figure 3.2: Fraction of bandwidth allocated to thread 2 as a function of miss frequency ratio of thread 2 to thread 1.

between the two applications diverge, the natural bandwidth share vs. optimum bandwidth partition diverge as well, especially when the miss frequencies differ a lot (e.g., $\frac{M_2A_2}{M_1A_1} < 0.5$ or $\frac{M_2A_2}{M_1A_1} > 2$). The figure shows that an application with a larger miss frequency tends to obtain a larger bandwidth share when no partitioning is applied, compared to the optimum bandwidth partition size, leading to the following observation:

Observation 4 *Compared to unmanaged bandwidth usage, the optimum bandwidth partitioning tends to slightly (but not overly) constrain applications with high miss frequencies as if to prevent them from dominating the bandwidth usage and starving the applications with lower miss frequencies.*

This observation obtained from our analytical model gives a theoretical foundation for qualitative empirical observation made in prior studies [64, 65, 66, 72].

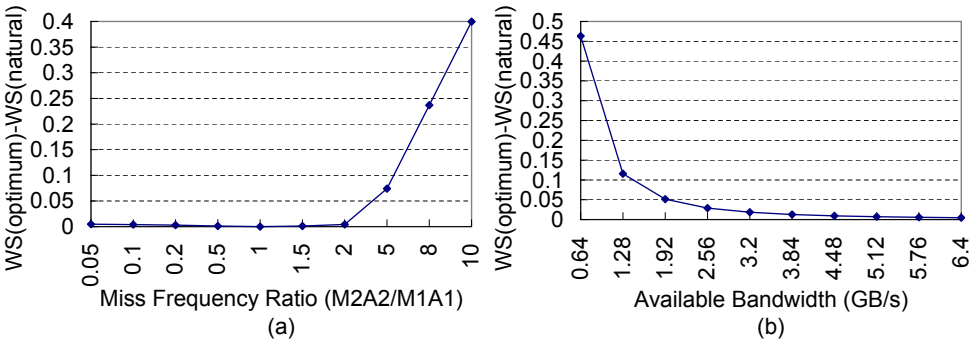


Figure 3.3: The weighted speedup when optimum bandwidth partition is made minus that when no bandwidth partitioning is used, as a function of the ratio of miss frequencies of thread 2 and thread 1 (a), or as a function of the peak available bandwidth B (b). For part (a), we assume $M_1A_1 = 2$million/s and $B = 1.6$GB/s. For part (b), we assume $\frac{M_2A_2}{M_1A_1} = 5$.

Figure 3.2 shows that the difference between the optimum bandwidth partition and the natural bandwidth share is never huge based on our assumption (within 10% from each other). How much can this seemingly minor difference in bandwidth allocation make a difference in system performance? To investigate this, we substitute $\beta_{2(optimum)}$ and $1 - \beta_{2(optimum)}$ into Equation 3.15 for a two-core system to get the weighted speedup of the system under optimum bandwidth partition. We further assume that cache block size $K = 64$ bytes, miss frequencies (M_1A_1) of 2 million/s, and total off-chip bandwidth $B = 1.6$GB/s. We also substitute $\beta_{2(natural)}$ and $1 - \beta_{2(natural)}$ into Equation 3.15 to get the weighted speedup when no bandwidth partitioning is applied. Subtracting these two weighted speedups, we obtain the difference between the weighted speedup under optimum partitioning vs. when no bandwidth partitioning is employed, as shown in Figure 3.3(a). The x-axes of the figure shows the miss frequency ratio of thread 2 to thread 1, obtained by increasing the miss frequency of thread 2 as we go to the right along the x-axes.

The first observation we can make from the figure is that as the miss frequency difference between threads increases, the higher the weighted speedup difference between optimum bandwidth partitioning and natural bandwidth share. This is evident in that when we go to the left or the right from the middle point ($\frac{M_2A_2}{M_1A_1} = 1$), the weighted speedup difference increases. However, the increase in the weighted speedup difference is asymmetric. It is relatively low as we go to the left from $\frac{M_2A_2}{M_1A_1} = 1$, but increases rapidly as we go to the right from $\frac{M_2A_2}{M_1A_1} = 1$. This asymmetry is interesting considering that the difference in bandwidth share between optimum partitioning and unmanaged bandwidth usage increases in both directions in a more symmetrical way in Figure 3.2. What this tells us is that scarcer bandwidth (due to high miss frequency of thread 2 on the right side of the figure) magnifies the impact of optimum bandwidth partitioning on system performance.

To make this observation clearer, Figure 3.3(b) shows the difference of weighted speedup between optimum and no bandwidth partitioning as affected by available peak bandwidth. We assume $\frac{M_2A_2}{M_1A_1} = 5$, while varying B from 0.64GB/s to 6.4GB/s (other parameters remain the same). The figure shows while the difference in miss frequency ratio of two threads is constant, its effect on system performance is magnified when bandwidth becomes scarcer, as we go to the left in the figure. To summarize, we arrive at the following observation:

Observation 5 *Whether bandwidth partitioning can improve system performance is affected primarily by the difference in miss frequencies between threads. However, the magnitude of this improvement is highly dependent on the scarcity of bandwidth, i.e. larger when bandwidth is scarcer.*

With the trend of increasing number of cores on a chip, the off-chip bandwidth is going to be increasingly shared by more cores. Thus, the impact of bandwidth partitioning on system performance will likely be higher in the future.

3.2.1 Interaction Between Cache and Bandwidth Partitioning

Cache partitioning could dynamically divide the shared cache capacity into portions allocated to each core in order to minimize cache misses, optimize weighted speedup or fairness, etc. We assume here that cache partitioning attempts to reduce the total number of L2 cache misses among all cores.

From the model, we can deduce that there are two ways by which cache and bandwidth partitioning interact. First, since cache partitioning reduces the total number of cache misses, it has the effect of relieving certain off-chip bandwidth contention. As shown in Figure 3.3(b), more abundant bandwidth results in smaller system performance improvement due to bandwidth partitioning. Second, cache partitioning may change miss frequency difference between two threads. If cache partitioning results in increasing the miss frequency difference between two threads, then the impact of bandwidth partitioning on system performance increases. Otherwise, the impact of bandwidth partitioning on system performance decreases. This leads us to the next observation:

Observation 6 *Cache partitioning may reduce the impact of bandwidth partitioning on system performance by (1) reducing the bandwidth pressure through reducing the total number of cache misses and (2) reducing the difference in miss frequencies between threads. On the other hand, cache partitioning may increase the impact of bandwidth partitioning on system performance by increasing the difference in miss frequencies between threads.*

The magnitude of the two interaction factors is difficult to predict using an analytical model, so we use empirical evaluation to quantify them in Section 3.3.

3.3 Model Validation and Empirical Evaluation

In the previous two sections, we have discussed how we construct our analytical model and how we use it to evaluate the impact of bandwidth partitioning on system performance. In this section, we show model validation, and empirical results that confirm the observations we obtained from the model.

3.3.1 Evaluation Environment and Methodology

Simulation Parameters. We use a cycle-accurate full system simulator based on Simics [60] to model a 4-core CMP. Each core has a scalar in-order issue pipeline with a 3.2GHz clock frequency. Each core has a private L1 instruction cache and L1 data cache with 16KB size, 2-way associativity, and 2-cycle access latency. The L2 cache is shared among four cores, and has 2MB size, 16-way associativity, and 8-cycle access latency. All caches have a block size of 64-byte, implement write-back policy, and use the LRU replacement policy. We do not assume an off-chip cache, but note that an off-chip cache will increase the importance

of bandwidth partitioning since uncontended memory access time declines with an off-chip cache, but contention-induced queuing delay does not. The bus to off-chip memory is a split transaction bus, with a peak bandwidth varied from 1.6GB/s to 25.6GB/s (with a base case of 6.4 GByte/s, in line with a DDR2-800 SDRAM channel [62]). The main memory is 2GB with 240-cycle access latency. The simulator ignores the impact of page mapping by assuming each application is allocated contiguous physical memory. The CMP runs Linux OS that comes with Fedora Core 4 distribution.

Cache Partitioning Method. For cache partitioning, we implement the per-set cache partitioning algorithm from [26, 36, 68] with the goal of minimizing the total number of L2 cache misses. At each 1-million cycle interval, the stack distance profile [77] (or marginal counters [83, 84]) of each thread is collected. The stack distance profile tells how many cache misses would have been eliminated if the thread had one more cache way, or how many cache misses would have been added if the thread had one less cache way. Based on this profile, for the next interval, we reallocate one cache way from the thread that is projected to increase the fewest number of cache misses if one cache way is taken, to the thread that is projected to decrease the largest number of cache misses if one cache way is added. This affects two threads, and the other two threads' partitions are unchanged.

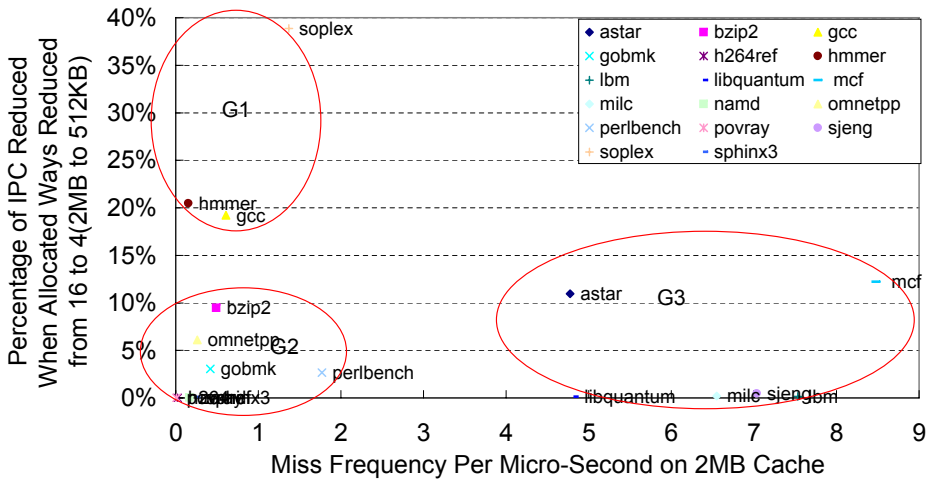


Figure 3.4: Benchmarks sensitivity to L2 cache capacity and miss frequency.

Workload Construction. We consider seventeen C/C++ benchmarks from the SPEC2006 benchmark suite [82]. Benchmarks written in Fortran were skipped due to compiler infrastructure limitations. We compile the benchmarks using `gcc` compiler with `O1` optimization level

into `x86` binaries. We use the *ref* input sets, simulate 250 million instructions after skipping the initialization phase of each benchmark, by manually inserting breakpoints at the point where major data structures have been initialized. We pair the benchmarks into co-schedules consisting of four benchmarks each.

To reduce the number of co-schedules that we need to evaluate, we categorize the benchmarks based on two criteria: sensitivity to L2 cache capacity (measured as the reduction of IPC when the L2 cache capacity is reduced from 2MB to 512KB), and off-chip bandwidth intensity (measured as the miss frequency when the benchmark runs alone in a system with peak off-chip bandwidth of 6.4GB/s). Figure 3.4 shows where each benchmark maps. From the figure, we can classify the benchmarks into three rough groups: G1 benchmarks are cache capacity sensitive but not bandwidth intensive, G3 benchmarks is cache capacity insensitive but bandwidth intensive, and G2 benchmarks are neither cache capacity sensitive nor bandwidth intensive. We choose three types of mixed workloads with each type containing two representative combinations taken from the benchmark groups (Table 3.3), based on the likelihood that cache or bandwidth partitioning will improve system performance. Mix-1-* workloads are expected to benefit from both cache and bandwidth partitioning as they contain cache capacity sensitive and bandwidth intensive benchmarks. Mix-2-* workloads are not expected to benefit from bandwidth partitioning because the miss frequencies of the benchmarks are almost identical. Mix-3-* workloads are expected to benefit from bandwidth partitioning since they contain bandwidth intensive benchmarks.

Table 3.3: Three representative cases of mixed workloads.

Workload types	Coscheduled Benchmarks	Expected Speedup	
		Cache	BW
Mix-1-1	*libquantum-milc-hmmer-gcc* (2G3+2G1)	Yes	Yes
Mix-1-2	*libquantum-libquantum-soplex-bzip2* (2G3+G1+G2)	Yes	Yes
Mix-2-1	*astar-libquantum-libquantum-libquantum* (4G3)	Maybe	No
Mix-2-2	*gobmk-gobmk-bzip2-bzip2* (4G2)	Maybe	Maybe
Mix-3-1	*mcf-milc-lbm-sjeng* (4G3)	Maybe	Yes
Mix-3-2	*libquantum-libquantum-milc-lbm*(4G3)	No	Yes

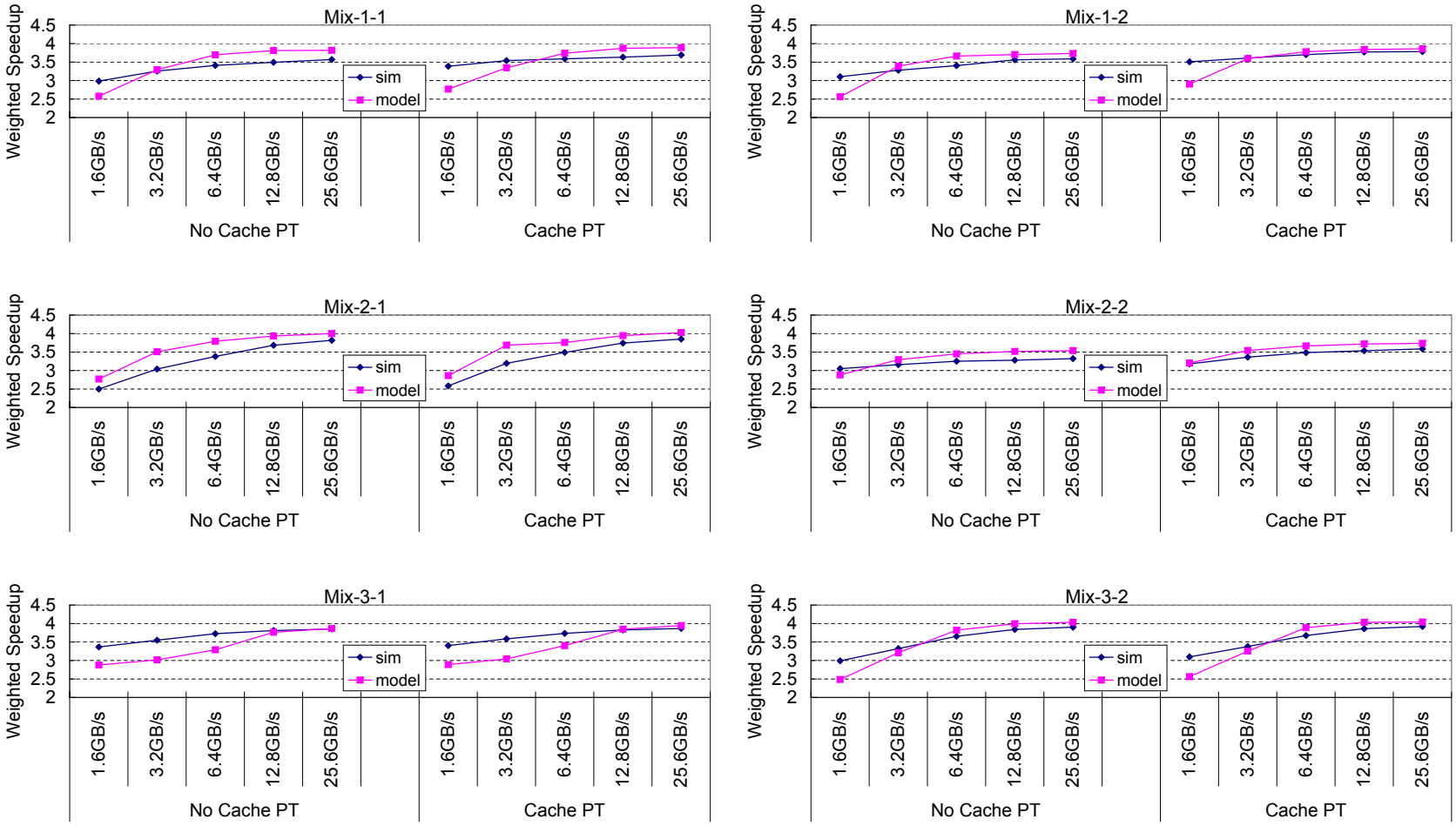


Figure 3.5: Comparing weighted speedup of simulated vs. modeled for mixed workloads.

3.3.2 Experimental Results

Model Validation

In this section, we investigate how well Equation 3.15 derived from our model predicts system performance. To provide the input necessary for the equation, we run each application alone and collect its CPI (CPI_{alone}) and average number of L2 cache misses per instruction (h_m). We also run each application with an infinite L2 cache size and collect its CPI ($CPI_{L2\infty}$). The effective L2 miss penalty (t_m) is computed from Equation 3.2. When applications run as a co-schedule, we apply near optimum bandwidth partitioning for each 1-million cycle interval (details of algorithm and parameters are shown in Section 3.3.2). At the end of each interval, we collect each application's miss frequency ($M_i A_i$) and bandwidth share (β_i). This per-interval data is input to the model to compute the predicted weighted speedup for that interval. The final weighted speedup is computed as the average over all intervals. We perform the predictions for when cache partitioning is applied and also when it is not applied.

Figure 3.5 shows the weighted speedup collected by the simulation (*sim*) and as estimated by our model (*model*), with the available peak off-chip bandwidth varied from 1.6GB/s to 25.6GB/s. The figure shows that the overall trends of the predicted weighted speedups fit well with the measured weighted speedups.

To quantify the prediction errors, we take the absolute value of the difference between predicted weighted speedup and measured weighted speedup, divided by the measured weighted speedup. Larger prediction errors generally occur at the lowest peak bandwidth (1.6GB/s), with the maximum prediction error occurs in Mix-1-1 (18.3% error). The arithmetic and geometric mean of the errors are low, 7.2% and 5.2%, respectively. We suspect the errors are not caused by the model, but instead by the inaccuracy of the inputs of the model, i.e. by taking CPI_{alone}, $CPI_{L2\infty}$ and t_m over the entire execution, while they might vary due to different phases. Another potential source of error is that bandwidth partitions are assumed to sum up to 100% in our prediction, while in reality the sum may be lower than 100%. This factor tends to make the predicted weighted speedup higher than the measured weighted speedup when the peak bandwidth is high.

Approximating the Optimum Bandwidth Partitioning

Our analytical model shows the criteria for weighted speedup-optimum bandwidth partition in Equation 3.18. However, the equation cannot be implemented in the real world, as the inputs to the equation, such as $CPI_{alone,i}$ and $CPI_{L2\infty,i}$, are not collectible. For a practical implementation, we have to approximate the partitions by estimating the CPIs. As we discussed in Section 3.1.2, miss frequencies are a much more important factor than the CPIs. Hence, in our simplified implementation, we simply ignore the CPIs and assume $\frac{CPI_{alone,i}}{CPI_{L2\infty,i}} = 1$. While no

longer optimum, we will show that the simplified optimum partitioning performs quite close to the optimum bandwidth partitioning, and outperforms other partitioning schemes.

Once bandwidth partitions are determined, the partitions are enforced by the token bucket algorithm [85] as discussed in Section 3.1.1). In order to adapt to changes in program (phases) behavior across time, we divide the execution into 1-million cycle intervals. At the end of each interval, we use the miss frequencies of co-scheduled threads collected in the interval that just ended, and readjust the bandwidth partitions for the next interval. While we assume unlimited token bucket size and perfect (100%) bandwidth utilization in the analytical model, the bucket size cannot be infinite in a practical implementation. The essential reason is that in the real world, memory bandwidth utilization may not achieve 100%, even if the available bandwidth is quite limited such as 1.6GB/s, due to the bursty nature of bandwidth usage. This implies that if the bucket size is unlimited, there are always unused tokens accumulated in the bucket over time. Since the demand (real bandwidth utilization) is much less than supply (100% token rates), eventually there are a large number of tokens piled up for each core. This makes the token bucket algorithm (which is supposed to throttle the memory requests from bandwidth-hungry core and accelerate memory requests from other cores) useless, because in this case memory requests from each core can always grab their required number of tokens in the infinite token bucket, and directly go to the off-chip interface, making it the same as natural sharing and lose the throttle feature.

The general approach to resolve this problem is to limit the bucket size for each core. However, naively choosing threshold for the bucket size may cause two problems. One problem is if the bucket size is too small, the burstiness of memory requests cannot be tolerated well. During the idle period, the small bucket is filled very fast and lots of tokens generated are lost due to the full bucket. When the period of bursty memory requests comes, the small bucket is drained quickly by each core and all cores have to wait for new tokens to be generated. Experiments show that too small bucket size can severely hurt system performance. The other problem is that due to the unknown number of lost tokens in limited-size buckets, the allocated bandwidth fraction to each core may not be the same as the optimum bandwidth partition sizes obtained from the model.

To solve both issues, we choose a global threshold for the total size of all buckets and dynamically reset the bucket size of each core according to the optimum bandwidth partitions. The global bucket size is obtained from empirical evaluations and it can guarantee to maintain the ability of tolerating burstiness of memory requests. Specifically, during each evaluation interval of 1-million cycles, at most 1-million tokens can be generated. For 1.6GB/s available bandwidth, we assume 90% average bandwidth utilization, hence the global threshold for 1.6GB/s is 0.9 million tokens. This average bandwidth utilization is decreased linearly with the increment of available bandwidth (e.g., 45% utilization and 0.45 million tokens per interval for

3.2GB/s, etc). Dynamically resetting the bucket size for each core according to the optimum partitions can guarantee that the number of lost tokens for each core also follows the optimum partitioning sizes, therefore the allocated bandwidth fraction to each core is the same as its optimum partitioning size.

To test our bandwidth partitioning algorithm, Figure 3.6 shows evaluation results for Mix-1-2, Mix-2-2 and Mix-3-2 workloads (results for the other combination of each Mix type show the same patterns). The figure shows weighted speedups of various schemes: no partitioning, fair partitioning which divides the off-chip bandwidth equally among cores (similar to Nesbit et al.'s fair queuing [66]), simplified optimum partitioning which assumes $\frac{CPI_{alone,i}}{CPI_{L2\infty,i}} = 1$, 3-pass optimum partitioning using the actual values of $CPI_{alone,i}$ and $CPI_{L2\infty,i}$ collected from two profiling passes, and partitions adjusted by 3%, 5%, 8% and 10% from the simplified optimum criteria. Specifically, for each adjustment amount, we select two applications and bump up their bandwidth partition by x%, correspondingly reduce the partition of the other two by x%. Note that there are $\binom{4}{2} = 6$ different runs for each x% adjustment case. We take the average of the 6 runs for each co-schedule and show the range (max and min of weighted speedup) as the candles.

The figure shows that as in prior studies [8, 33, 64, 65, 66, 72], fair partitioning improves system performance, especially for cases in which the peak off-chip bandwidth is low, mainly because it prevents applications with high miss frequencies from dominating the bandwidth usage. However, our simplified bandwidth partitioning algorithm outperforms fair partitioning in all cases. The improvement in weighted speedups over no partitioning due to simplified optimum partitioning are between 20% to 370% higher compared to the improvement in weighted speedups due to fair partitioning (129% on average). This indicates that simplified partitioning is significantly more effective than fair partitioning. Compared to the 3-pass optimum partitioning, the simplified optimum partitioning under-performs slightly across all cases, indicating that assuming $\frac{CPI_{alone,i}}{CPI_{L2\infty,i}} = 1$ does not hurt performance by much. Compared to various adjusted bandwidth cases, even the upper end in small adjustments (3% and 5%) at best match the performance achieved by our simplified partitioning, indicating that our simplified optimum partitioning already chooses partition sizes that are less than 3% away from those chosen by the 3-pass optimum partitioning, and adjusting them by a larger amount decreases performance, to the extent that larger adjustments (8% and 10%) perform even worse than no partitioning in many cases. The reason is that arbitrarily partitioning the available bandwidth may cause bandwidth fragmentation, because the memory requests that should have been served on the idle bus now have to wait for new tokens, while other applications that currently have tokens do not have memory requests to utilize them.

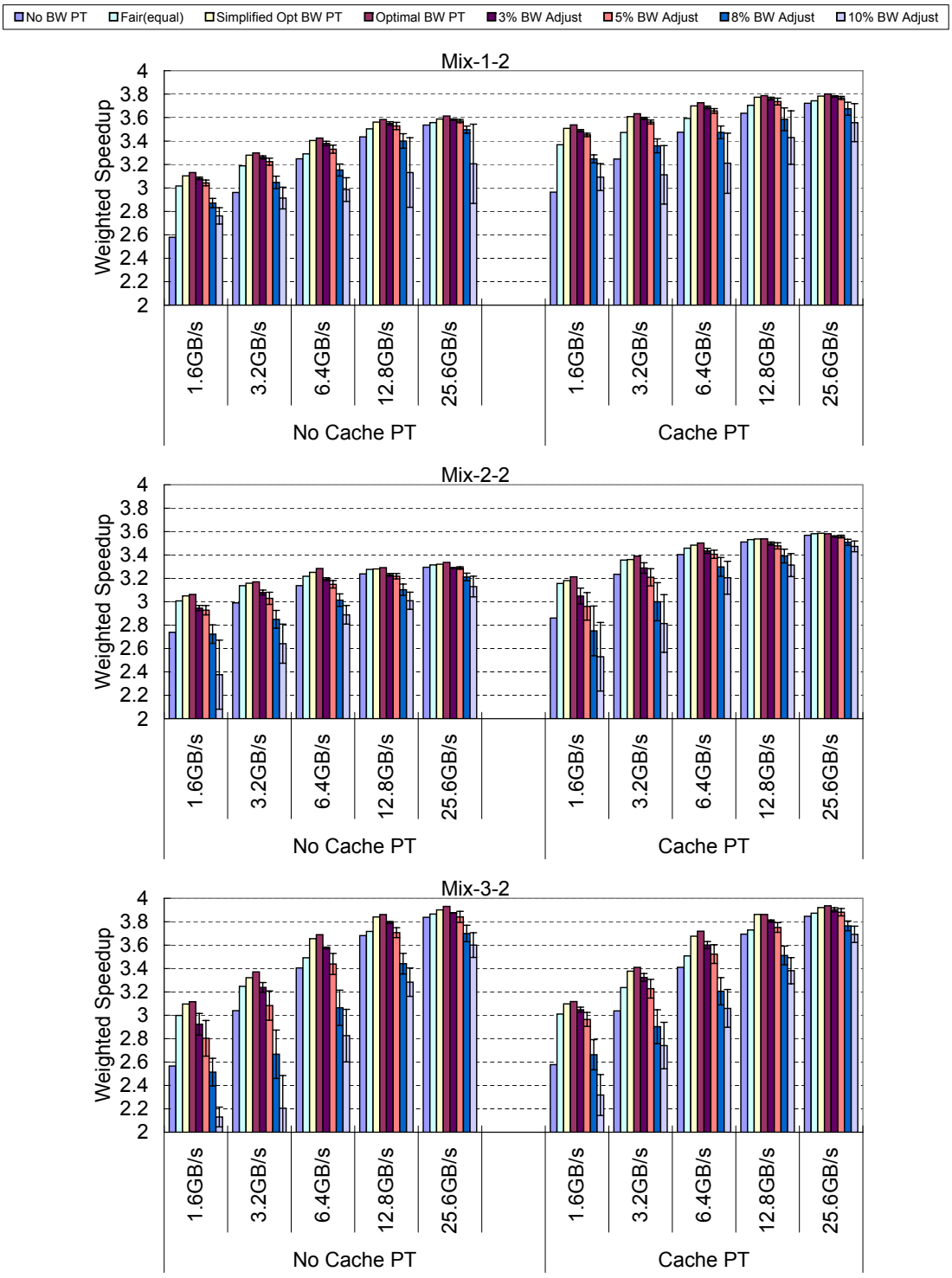


Figure 3.6: Comparing the weighted speedup for Mix-1-2, Mix-2-2 and Mix-3-2 using fair (equal) partitioning [66] vs. simplified optimum bandwidth partitioning vs. optimum partition and with various adjustments.

Mix-1 Results

We test four configurations: no cache or bandwidth partitioning (No PT), only cache partitioning with a goal of minimizing the total number of cache misses (Cache PT), only simplified bandwidth partitioning from Section 3.3.2 (BW PT), and both cache and bandwidth partitioning (Both PT). To investigate the interaction between cache and bandwidth partitioning, we use Alameldeen et al.'s approach for deducing the magnitude of interaction between two factors [6]. Specifically, we express the combined speedup of cache partitioning (cachePT) and bandwidth partitioning (bwPT) as $Spdup(cachePT, bwPT) = Spdup(cachePT) \times Spdup(bwPT) \times (1 + Interaction)$. By computing $Interaction$, we can deduce whether they positively or negatively impact one another.

The weighted speedups of each co-schedule for the Mix-1 workloads are shown in Figure 3.7. Table 3.4 shows the weighted-speedup ratios of cache partitioning and bandwidth partitioning over unmanaged cache and bandwidth usage (No PT), the interaction factor, and coefficient of variation (CV) of the miss frequencies of co-scheduled threads. From the figure and the table, we can make several observations.

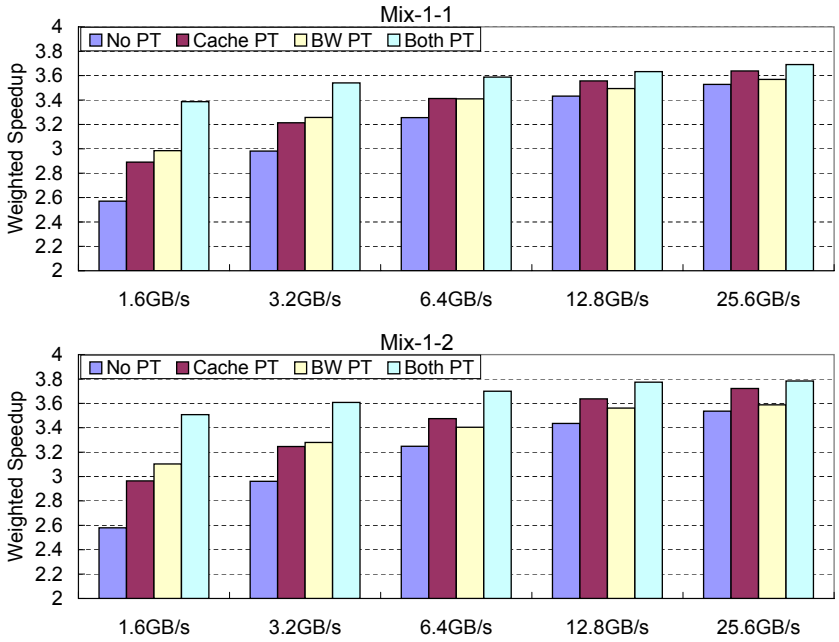


Figure 3.7: Weighted speedup of four configurations with various available bandwidth for Mix-1 workloads.

First, contrary to a theory in literature [12, 84], bandwidth partitioning's impact on system

Table 3.4: Weighted speedup ratios over no partitioning configuration and interaction of cache and bandwidth partitioning for Mix-1 workloads.

Workloads	BW	Weighted Speedup ratios over NoPT			Interact	CV of Miss Freq	
		Cache PT	BW PT	Both PT		No PT	Cache PT
Mix-1-1	1.6GB/s	1.124	1.161	1.317	0.9%	0.418	0.625
	3.2GB/s	1.078	1.092	1.188	0.8%	0.435	0.657
	6.4GB/s	1.048	1.047	1.102	0.4%	0.456	0.660
	12.8GB/s	1.036	1.018	1.058	0.3%	0.489	0.666
	25.6GB/s	1.031	1.012	1.046	0.3%	0.487	0.671
Mix-1-2	1.6GB/s	1.149	1.203	1.360	-1.6%	0.309	0.454
	3.2GB/s	1.096	1.107	1.218	0.3%	0.323	0.494
	6.4GB/s	1.070	1.048	1.139	1.6%	0.334	0.519
	12.8GB/s	1.059	1.037	1.098	0.1%	0.344	0.543
	25.6GB/s	1.052	1.014	1.070	0.2%	0.349	0.557

performance is not necessarily secondary in comparison to cache partitioning. In fact, for the low peak bandwidth cases (1.6 GB/s and 3.2 GB/s), bandwidth partitioning improves weighted speedup ratios by more than cache partitioning (16% vs. 12% and 9% vs. 8% in Mix-1-1, 20% vs. 15% and 11% vs. 10% in Mix-1-2). For higher peak bandwidth cases, its weighted speedup ratios are lower than that of cache partitioning, but they are still significant.

Secondly, for the bandwidth partitioning-only case, we can see that the improvement of weighted speedup diminishes as peak bandwidth grows (16.1% for 1.6GB/s down to 1.2% for 25.6GB/s in Mix-1-1, 20.3% for 1.6 GB/s down to 1.4% for 25.6 GB/s in Mix-1-2). This agrees with Observation 5 and Figure 3.3 in Section 3.2. The reason is explained by our model: the level of contention, and hence the queuing delay suffered by memory requests, declines as peak bandwidth grows. Thus, the magnitude of performance improvement due to bandwidth partitioning declines. However, we cannot take it as good news because if the number of cores integrated on a single chip grows faster than the off-chip bandwidth in the future, the amount of off-chip bandwidth available per core declines.

Thirdly, the interaction factors between cache and bandwidth partitioning in Table 3.4 are positive (except on 1.6 GB/s in Mix-1-2), indicating a prevailing synergistic interaction between cache and bandwidth partitioning. As discussed in Section 3.2.1, the interaction will tend to be negative because cache partitioning reduces the total number of cache misses and hence the off-chip bandwidth pressure, unless there is an offsetting effect of increased difference in miss frequencies which causes bandwidth partitioning's impact on system performance to increase. In fact, this is exactly what happens. The coefficient of variation (CV) of miss frequencies, which measures the variability of miss frequencies among co-scheduled threads, increase significantly when cache partitioning is applied: in Mix-1-1, CVs increase from 0.4's to 0.6's, whereas in Mix-1-2, CVs increase from 0.3's to 0.4-0.5s. The increase in CVs improves the impact of bandwidth

partitioning on weighted speedup, to a degree large enough to produce a synergistic interaction between cache and bandwidth partitioning (except for the 1.6GB/s case in Mix-1-2).

Mix-2 Results

For Mix-2 workloads, we expect that the performance improvement from bandwidth partitioning to be much smaller than in Mix-1 workloads (due to the benchmarks having similar miss frequencies). However, the benchmarks may still benefit much from cache partitioning because of the difference in sensitivities to cache space. Figure 3.8 shows the weighted speedups when various partitioning schemes are applied for Mix-2 workloads.

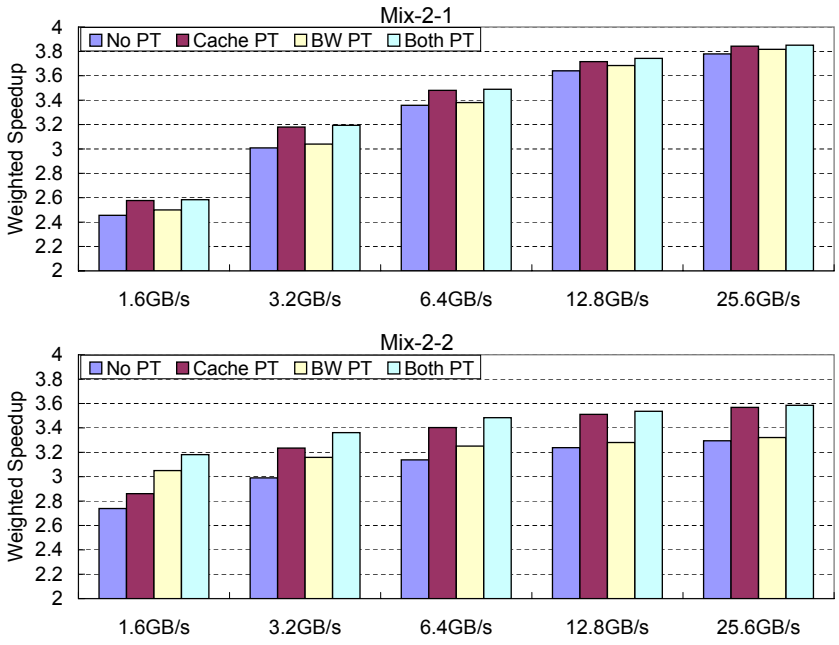


Figure 3.8: Weighted speedup of four configurations with various available bandwidth for Mix-2 workloads.

Overall, Mix-2-1 shows little performance improvement from bandwidth partitioning (speedups ranging from 1-2%). This is expected because according to our model, applications having similar miss frequencies achieve natural bandwidth shares that are already near optimum partitions. CPI_{alone} and $CPI_{L2\infty}$ play a little role here because they are almost equal. Mix-2-2, however, enjoys a good performance improvement from bandwidth partitioning (up to 14%). Upon a closer inspection, we found that while the benchmarks have similar miss frequencies when running alone, when they are co-scheduled, the difference in miss frequencies increases, with *bzip2*'s miss frequency becoming 3× of that of *gobmk*. Cache partitioning, on the other hand, improves

system performance by 2-6% for Mix-2-1 and 4-8% for Mix-2-2, mainly because *astar* and *bzip2*, are somewhat sensitive to cache capacity, being allocated more space than other benchmarks (Figure 3.4).

Table 3.5 shows that the interaction between cache partitioning and bandwidth partitioning is always negative, indicating a muted impact of bandwidth partitioning on system performance after cache partitioning is applied. Section 3.2 shows that this can happen due to reduced bandwidth pressure from fewer misses, or from reduced difference in miss frequencies among threads. The table 3.5 shows that the latter explanation is correct (CVs decline after cache partitioning is applied).

Table 3.5: Weighted speedup ratios over no partitioning configuration and interaction of cache and bandwidth partitioning for Mix-2 workloads.

Workloads	BW	Weighted Speedup ratios over NoPT			Interact	CV of Miss Freq	
		Cache PT	BW PT	Both PT		No PT	Cache PT
Mix-2-1	1.6GB/s	1.050	1.018	1.052	-1.5%	0.032	0.014
	3.2GB/s	1.057	1.010	1.061	-0.6%	0.011	0.004
	6.4GB/s	1.037	1.007	1.039	-0.4%	0.035	0.021
	12.8GB/s	1.021	1.011	1.028	-0.5%	0.034	0.012
	25.6GB/s	1.017	1.010	1.019	-0.8%	0.033	0.006
Mix-2-2	1.6GB/s	1.044	1.114	1.161	-0.2%	0.648	0.551
	3.2GB/s	1.081	1.056	1.234	-1.6%	0.710	0.550
	6.4GB/s	1.084	1.036	1.110	-1.2%	0.728	0.577
	12.8GB/s	1.084	1.013	1.092	-0.6%	0.757	0.589
	25.6GB/s	1.083	1.008	1.088	-0.3%	0.776	0.596

Mix-3 Results

For Mix-3 workloads, we expect a large benefit from bandwidth partitioning (due to large difference in miss frequencies for co-scheduled benchmarks), but there may be little benefit from cache partitioning (due to the benchmarks being insensitive to cache capacity). Figure 3.9 shows the weighted speedups when various partitioning schemes are applied for Mix-3 workloads.

The results largely match the expectation we derive from our model. The figure shows large performance improvement from bandwidth partitioning but little improvement from cache partitioning for both Mix-3-1 and Mix-3-2. The results confirm our model study (Figure 3.2 and 3.3) that (1) a large difference in miss frequencies causes the natural bandwidth share to deviate from the optimum bandwidth partitions, and (2) even a small difference can cause a large difference in system performance when the off-chip bandwidth is relatively scarce.

We also find that the interaction factor is negative for Mix-3-1 but positive for Mix-3-

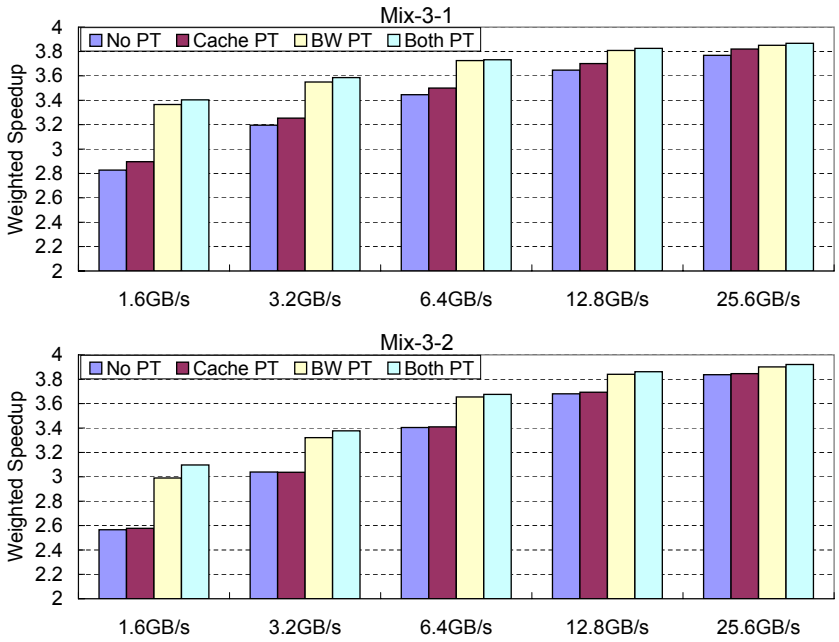


Figure 3.9: Weighted speedup of four configurations with various available bandwidth for Mix-3 workloads.

2. Again, the change in the CVs of miss frequencies after cache partitioning is applied is responsible for this, confirming Observation 6 of our model. Note that we can make a very interesting observation from the positive interaction for Mix-3-2:

Observation 7 *As Figure 3.9 illustrates, in some cases, although cache partitioning cannot improve performance, it can be useful to apply it in order to enhance the effectiveness of bandwidth partitioning in improving system performance.*

3.4 Related Work

Cache Partitioning. Many cache partitioning schemes have been proposed to target different optimization goals, such as the overall miss rate, fairness, overall IPC, or QoS. Suh et al. [83, 84] proposed to use marginal gain counters to guide cache partitioning in order to minimize the total number of cache misses. Qureshi and Patt [70] categorized workloads based on the sensitivity of IPC to cache capacity and proposed utility-based cache partitioning scheme to system performance. Kim et al. [77] proposed a cache partitioning scheme to guarantee a uniform slowdown among coscheduled threads. Nesbit et al. [68] proposed an abstraction of private cache on a shared cache substrate and used fair queuing algorithm to guide partitioning of the shared cache in order to achieve IPC comparable to that of using real private caches. Chang and Sohi [13] proposed a cooperative cache partitioning scheme to minimize the overall per-thread

Table 3.6: Weighted speedup ratios over no partitioning configuration and interaction of cache and bandwidth partitioning for Mix-3 workloads.

Workloads	BW	Weighted Speedup ratios over NoPT			Interact	CV of Miss Freq	
		Cache PT	BW PT	Both PT		No PT	Cache PT
Mix-3-1	1.6GB/s	1.024	1.190	1.204	-1.3%	0.554	0.546
	3.2GB/s	1.018	1.110	1.122	-0.8%	0.579	0.571
	6.4GB/s	1.016	1.081	1.083	-1.4%	0.613	0.586
	12.8GB/s	1.015	1.044	1.049	-1.0%	0.604	0.595
	25.6GB/s	1.014	1.022	1.026	-0.9%	0.611	0.600
Mix-3-2	1.6GB/s	1.004	1.165	1.207	3.1%	0.252	0.302
	3.2GB/s	0.999	1.093	1.111	1.7%	0.262	0.297
	6.4GB/s	1.002	1.074	1.080	0.4%	0.276	0.297
	12.8GB/s	1.004	1.043	1.049	0.2%	0.290	0.301
	25.6GB/s	1.002	1.017	1.022	0.3%	0.305	0.310

slowdown and maximize harmonic mean of per-thread IPC speedup over equal partitioning case. Hsu et al. [30] studied the impact of various performance metrics in guiding different cache partitioning policies. Iyer described the necessity of enabling QoS for threads that have different priorities [36], and proposed a QoS-aware cache and memory architecture [37]. Guo et al. [26] proposed a framework to provide QoS to applications with different requirements through cache partitioning techniques. Rafique et al. [71] proposed an architecture support that uses the OS interface to provide cache partitioning. Cho and Jin [14] proposed an OS-level page allocation algorithm in a shared non-uniform cache architecture to reduce cache access latency. Lin et al. [54] proposed a software approach to provide cache partitioning based on various optimization objectives. Jiang et al. [41] proposed hardware support and OS scheduling algorithm to maximize asymmetric cache performance in heterogeneous CMPs. None of these studies investigated how cache partitioning decisions will impact bandwidth partitioning.

Off-Chip Bandwidth Management and Partitioning. There are two categories of work in terms of managing the off-chip bandwidth usage: (1) memory controller scheduling policy for various types of memory requests, and (2) partitioning of off-chip bandwidth across different cores. Only the partitioning category is directly related to our work. In the first category (scheduling policy), Rixner et al. [73] proposed FR-FCFS scheduling policy to favor requests that hit in the row buffer over other requests, and older requests over younger ones within a timing constraint. Lee et al. [50] proposed Prefetch-Aware DRAM Controller (PADC) to keep the benefit of useful prefetches and avoid the harmful effect of useless prefetches. In the second category (partitioning), Nesbit et al. [66] proposed the Fair Queuing Memory Scheduler (FQMS) to prioritize memory requests according to the QoS objectives of various threads. FQMS achieves fair use of the off-chip bandwidth by enforcing each thread to use an equal fraction of the available bandwidth. However, it has a possible starvation and may suffer from

unexpected longer latencies. To avoid such a problem, Rafique et al. [72] improved FQMS with a heuristic to adaptively change the fraction of available bandwidth each thread could use, while Mutlu and Moscibroda proposed Stall Time Fair Memory Scheduler (STFM) to equalize the slowdown among equal-priority threads [64], and further improved STFM to exploit parallelism to group multiple requests from one thread as a scheduling unit [65]. Ipek et al. [33] proposed a reinforcement learning (RL) based memory controller that uses a machine learning approach to dynamically adapt scheduling decisions. Overall, these studies have shown that allocating an equal (fair) fraction of off-chip bandwidth partitions to all cores can improve the overall system performance. However, many questions are still unanswered, for example what are the fundamental factors affecting the effectiveness of bandwidth partitioning, can equal fractions achieve the best system performance, and how do cache partitioning and bandwidth partitioning interact?

Coordinated Cache and Bandwidth Partitioning. Bitirgen et al. [8] proposed a global resource manager that uses Artificial Neural Networks (ANNs) to manage the allocation of shared cache capacity and off-chip bandwidth. The study showed that coordinated cache and bandwidth partitioning can outperform unmanaged cache and bandwidth partitioning. As ANN is a black box optimizer, fundamental insights of how cache and bandwidth partitioning interact remain undiscovered.

3.5 Conclusions

The goal of this chapter is to understand how off-chip bandwidth partitioning affects system performance, and how cache and bandwidth partitioning interact. We have presented a simple yet powerful analytical model that we believe has achieved the goal. Constructing and studying the model lead us to several subtle, yet surprising, observations. For example, we disproved a theory in literature that bandwidth partitioning is of secondary importance compared to cache partitioning. We found that the scarcer off-chip bandwidth becomes, the more bandwidth partitioning can improve system performance significantly. Bandwidth partitioning can only improve system performance when miss frequencies between threads differ significantly. Optimum bandwidth partitions can be derived theoretically, and can be implemented in practice with some simplifications. Cache partitioning and bandwidth partitioning can sometimes produce synergistic interaction. This produces an interesting scenario in which even cache partitioning alone does not improve performance in some cases, it can act as a *performance booster* for bandwidth partitioning.

Chapter 4

Impact of Hardware Prefetching and Bandwidth Partitioning

This chapter is organized as follows: Section 4.1 shows the construction of the analytical model, Section 4.2 discusses model based exploration, Section 4.3 validates the findings obtained from the model through simulation based evaluation, Section 4.4 reviews the related work, and Section 4.5 concludes the insights in this chapter [58].

4.1 Analytical Modeling

In this section, we discuss the assumptions used by our model (Section 4.1.1), derive a metric that determines whether prefetching benefits performance (Section 4.1.2), and present our prefetching and bandwidth partitioning model (Section 4.1.3).

4.1.1 Assumptions and Model Parameters

Assumptions. This study assumes a CMP with homogeneous cores, where each core has private L1 (instruction and data) and L2 (unified) caches. Each L2 cache has a hardware prefetcher that can be turned on or off, similar to the one used in IBM Power processors [7, 29, 32]. The off-chip bandwidth is shared by all cores through a single queue interface, where all off-chip memory requests are served in First-Ready First-Come-First-Serve (FR-FCFS) policy [73], a common base implementation in memory controllers. Our study focuses on multi-programmed workloads, hence we assume that single-threaded applications run on different cores and do not share data.

As in Chapter 3, we define bandwidth partitioning as allocating fractions of off-chip bandwidth to different cores and enforcing these fractions as per-core quota. Thus, bandwidth partitioning reduces inter-core interference, without changing the underlying memory access scheduling policy. The bandwidth partitioning is implemented using token bucket algorithm [57], with the goal to optimize the system throughput expressed as weighted speedup as shown in Equa-

tion 3.1. In this model, we take into account the off-chip memory requests due to both L2 cache misses and prefetches, but ignore the memory traffic such as write backs that are not on the critical path of performance, as well as coherence messages since we consider applications that do not share data.

Model Parameters. The input to the model includes system parameters as well as per-thread parameters is listed in Table 4.1.

Table 4.1: Input and parameters used in our model.

System parameters	
f	CPU clock frequency (Hz)
K	Cache block size (Bytes)
B	Peak off-chip memory bandwidth (Bytes/sec)
N	Number of cores in the CMP
Thread-specific parameters	
$CPI_{L2\infty,i}$	Thread i's CPI assuming infinite L2 cache size
$CPI_{alone,i}$	Thread i's CPI when it runs alone in the CMP
M_i	Thread i's L2 cache miss rate without prefetching
A_i	Thread i's L2 cache access frequency (#accesses per second)
$M_{p,i}$	Thread i's L2 cache miss rate after prefetching
P_i	Thread i's L2 prefetching rate (ratio of #prefetches to #L2 accesses)
c_i	Thread i's prefetching coverage
a_i	Thread i's prefetching accuracy
$T_{m,i}$	Thread i's average memory access latency (#cycles)
$\beta_{p,i}$	Fraction of off-chip bandwidth assigned to thread i with prefetching

4.1.2 Composite Metric for Prefetching

Traditional metrics for prefetching performance include *coverage*, defined as the fraction of the original cache misses that are eliminated by prefetching; and *accuracy*, defined as the fraction of prefetch requests that successfully eliminate cache misses. These metrics cannot determine whether prefetching should be used or not in limited off-chip bandwidth environment because they do not take into account how much off-chip bandwidth is available, and how memory access latency is affected by prefetching.

To arrive at a new metric that can be used to determine whether prefetching is profitable for performance, taking into account the amount of available bandwidth, we start from the basic CPI model as shown in Equation 3.2 and add the effect of additional queueing delay due to prefetching traffic. We use Δt to represent the queueing delay on the bus and let T_m be the

average access latency to the memory, [1] the CPI becomes:

$$CPI = CPI_{L2\infty} + h_m \cdot (T_m + \Delta t) \tag{4.1}$$

When prefetching is applied in the system, let CPI_p present the new CPI, Δt_p as the new queuing delay on the bus, $h_{m,p}$ as the new L2 misses per instruction, the CPI equation then becomes:

$$CPI_p = CPI_{L2\infty} + h_{m,p} \cdot (T_m + \Delta t_p) \tag{4.2}$$

Let us examine how hardware prefetching affects CPI (Equation 4.2 vs. Equation 4.1). Prefetching may decrease the L2 miss per instruction, i.e. $h_{m,p} < h_m$, but increase the queueing delay, i.e. $\Delta t_p > \Delta t$ due to the extra traffic generated by prefetch requests.

Note that $h_{m,p}$ can be expressed as the multiplication of miss frequency $(M_p \cdot A_p)$ and average time taken to execute one instruction $(\frac{CPI_p}{f})$:

$$h_{m,p} = \frac{M_p \cdot A_p \cdot CPI_p}{f} \tag{4.3}$$

Substituting Equation 4.3 into Equation 4.2, and solving CPI_p, we can arrive at:

$$CPI_p = \frac{CPI_{L2\infty}}{1 - \frac{M_p A_p}{f}(T_m + \Delta t_p)} \tag{4.4}$$

Now we will derive Δt_p using Little's law. If we let λ_p denote the arrival rate of memory requests, then λ_p is equal to the sum of frequency of cache miss and prefetch requests:

$$\lambda_p = M_p \cdot A_p + P \cdot A_p = (M_p + P)A_p \tag{4.5}$$

Little's law [55] for a queuing system states that the average queue length (L_p) is equal to the arrival rate (λ_p) multiplied by the service time (T_p), i.e. $L_p = \lambda_p \cdot T_p$, while the service time on the off-chip bus is the cache block size (K) divided by the available bandwidth (B). Hence, the average number of memory requests arriving during the service time is:

$$L_p = \frac{K}{B} \cdot (M_p + P)A_p \tag{4.6}$$

If we assume that memory requests are not bursty, i.e. requests are processed back to back, the average waiting time of a newly arriving request is equal to the L_p requests that are ahead of it in the queue, multiplied by the service time. Thus, the waiting time Δt_p (in cycles) can

[1] The average memory access latency T_m is an amortized value that implicitly includes the effect of memory bank and row buffer conflicts, Instruction Level Parallelism (ILP) as well as Thread Level Parallelism (TLP).

be expressed as:

$$\Delta t_p = f \cdot \frac{K}{B} \cdot L_p = f \cdot \frac{K^2}{B^2} \cdot (M_p + P)A_p \tag{4.7}$$

Substituting Equation 4.7 into Equation 4.4, expression for CPI_p is:

$$CPI_p = \frac{CPI_{L2\infty}}{1 - \frac{M_p A_p T_m}{f} - \frac{M_p(M_p+P)A_p^2 K^2}{B^2}} \tag{4.8}$$

Similarly, the system that does not employ prefetching has the CPI of:

$$CPI = \frac{CPI_{L2\infty}}{1 - \frac{MAT_m}{f} - \frac{M^2 A^2 K^2}{B^2}} \tag{4.9}$$

In order for prefetching to produce a net benefit in performance, the CPI after prefetching must be smaller than CPI without prefetching, i.e. $CPI_p < CPI$. From the definition of prefetching coverage and accuracy, we have (assuming L2 cache access frequency is not affected by prefetching, i.e. $A_p = A$):

$$c = \frac{MA - M_p A_p}{MA} = \frac{M - M_p}{M} \tag{4.10}$$

$$a = \frac{MA - M_p A_p}{PA_p} = \frac{M - M_p}{P} \tag{4.11}$$

Rearranging Equation 4.10 and Equation 4.11 to express M_p and P in terms of c and a, substituting them into the CPI_p expression in Equation 4.8, and simplifying the inequality $CPI_p < CPI$, we obtain the final expression for the inequality:

$$T_m > \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a}) \tag{4.12}$$

The right hand side of the inequality, $\frac{MAK^2}{B^2} f(c-2+\frac{1-c}{a})$, is the composite metric that takes into account prefetching coverage and accuracy, as well as cache block size, available bandwidth, and miss frequency. The left hand side of the inequality, T_m, measures the average exposed memory access latency. The inequality essentially provides a break-even point threshold for how large the average memory access latency should be for prefetching to be beneficial. It is easier to meet the inequality when the average memory access latency is large, the cache miss frequency is small, the available bandwidth is large, and the coverage and accuracy are large. All these factors make sense qualitatively, and Equation 4.12 captures their quantitative contributions. This leads us to:

Observation 8 *Whether prefetching improves or degrades performance cannot be measured just by its coverage or accuracy. Rather, a prefetching profitability criterion in Inequality 4.12 is needed, using input parameters that are relatively easy to collect in hardware.*

Recall that in the model construction for the average total waiting time, we have assumed

memory requests are not bursty and requests are contiguously processed back to back. This allows us to reach bandwidth utilization of 100% if the arrival rate of memory requests is high. However, in reality, due to the burstiness of memory requests, even a high arrival rate cannot achieve 100% bandwidth utilization. This introduces a small inaccuracy in our metric. Thus we can define a parameter $\alpha \in (0, 1)$ such that:

$$T_m > \frac{MAK^2}{\alpha B^2} f(c - 2 + \frac{1-c}{a}) \tag{4.13}$$

α loosely represents the degree of burstiness of memory requests and can be obtained from empirical evaluation (as shown in Section 4.3.2). Since $\alpha < 1$, the minimum value of the right hand side of the inequality is $\frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$. Thus we can conclude:

Observation 9 *If an application running on a core satisfies* $T_m < \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$, *prefetching is harmful to performance. In addition, if* $0 > \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$, *prefetching improves performance.*

4.1.3 Memory Bandwidth Partitioning Model with Prefetching

This model extends the bandwidth partitioning model from [57] by taking into account not just demand fetches, but also prefetch requests, and prefetching coverage and accuracy. In a CMP system, multiple threads running on different cores will be generating their own cache miss and prefetch requests. Requests from multiple cores will compete for the off-chip bandwidth, resulting in queuing delay to access the main memory. Since there are multiple threads running simultaneously, we will denote the CPI expression from Equation 4.4 for thread i as:

$$CPI_{p,i} = \frac{CPI_{L2\infty,i}}{1 - \frac{M_{p,i}A_i}{f}(T_{m,i} + \Delta t_{p,i})} \tag{4.14}$$

$\Delta t_{p,i}$ is the queuing delay suffered by thread i on the shared bus. With a similar derivation method in [57], we can compute it using Little's law for the case where we do not apply bandwidth partitioning (i.e. requests from all cores contend with each other for bandwidth access naturally) and compare it against the case where we apply bandwidth partitioning (i.e. bandwidth fraction is allocated to each core for its own requests). The CPI of thread i in a system without bandwidth partitioning can be expressed as:

$$CPI_{p,i,nopt} = \frac{CPI_{L2\infty,i}}{1 - \frac{M_{p,i}A_iT_{m,i}}{f} - \frac{\left(\sum_{j=1}^{N}(M_{p,j}+P_j)A_j\right)^2 M_{p,i}K^2}{(M_{p,i}+P_i)B^2}} \tag{4.15}$$

Let $\beta_{p,i}$ denote the fraction of bandwidth allocated to thread i. Using similar derivation,

the CPI for thread i for a system with bandwidth partitioning will be:

$$CPI_{p,i,bwpt} = \frac{CPI_{L2\infty,i}}{1 - \frac{M_{p,i}A_i \cdot T_{m,i}}{f} - \frac{M_{p,i}(M_{p,i}+P_i)A_i^2K^2}{\beta_{p,i}^2 \cdot B^2}} \tag{4.16}$$

Comparing Equation 4.16 and 4.15, we can conclude that Equation 4.15 is a special case of Equation 4.16 where:

$$\beta_{p,i} = \frac{(M_{p,i} + P_i)A_i}{\sum_{j=1}^{N}(M_{p,j} + P_j)A_j} \tag{4.17}$$

which leads us to the following observation:

Observation 10 *In a CMP system where off-chip bandwidth usage among multiple cores is unregulated, the off-chip bandwidth is naturally partitioned between cores, where the natural share of bandwidth a core uses is equal to the ratio of memory request frequency (including both cache misses and prefetch requests) of the core to the sum of all memory request frequencies from all cores.*

Substituting CPI from Equation 4.16 into Equation 3.1, we obtain:

$$WS_p = \sum_{i=1}^{N} C_i \left(1 - \frac{M_{p,i}A_iT_{m,i}}{f} - \frac{M_{p,i}(M_{p,i} + P_i)A_i^2K^2}{\beta_{p,i}^2B^2}\right) \tag{4.18}$$

where $C_i = \frac{CPI_{alone,i}}{CPI_{L2\infty,i}}$. Given that the first and second terms in the equation are not affected by bandwidth partitioning, to maximize weighted speedup, we need to minimize the sum of the third terms in the above equation, i.e.:

$$F(\beta_1, \ldots, \beta_N) = \sum_{i=1}^{N} C_i \frac{M_{p,i}(M_{p,i} + P_i)A_i^2K^2}{\beta_{p,i}^2B^2} \tag{4.19}$$

Minimizing $F(\beta_{p,1}, \beta_{p,2}, \ldots, \beta_{p,N})$ is a constrained optimization problem, with constraint of $\sum_{i=0}^{N} \beta_{p,i} = 1$. Solving via Lagrange multipliers [31], the bandwidth partition for thread i ($\beta_{p,i}$) that maximizes the weighted speedup is:

$$\beta_{p,i} = \frac{\left(C_iM_{p,i}(M_{p,i} + P_i)A_i^2\right)^{1/3}}{\sum_{j=1}^{N}\left(C_jM_{p,j}(M_{p,j} + P_j)A_j^2\right)^{1/3}} \tag{4.20}$$

leading us to the next observation:

Observation 11 *Weighted speedup-optimum bandwidth partition for a thread can be expressed as a function of all co-scheduled threads' miss and prefetch frequencies, infinite-L2 CPIs, and CPIs when each thread runs alone in the system, as in Equation 4.20.* [2]

[2]Note that the parameters in Equation 4.20 may vary due to the changes in an application's behavior throughout its execution. The equation does not assume them to be constant values, but assumes the parameters for any specific execution interval of the application are available.

4.2 Model-Driven Study

In this section, we will explore the bandwidth partitioning model derived in the previous section to gain insights into the how prefetching and bandwidth partitioning interact and affect system performance. To simplify the discussion, but without losing generality, we will assume dual-core CMP running two threads.

From Equation 4.20, we will assume that $C_i = \frac{CPI_{alone,i}}{CPI_{L2\infty,i}} = 1$. The reason is that C_i is a ratio of two CPIs, and the ratio tends to approach 1 in a system with many cores and a very large shared cache. Furthermore, bias of program behavior in terms of instruction level parallelism affects both the numerator and denominator. Hence ignoring C_i does not change the qualitative observations we will make in this section.

The model-driven study will be split into three cases depending on what factors are varied: miss frequency, prefetching coverage, and prefetching accuracy. For all cases, we assume available bandwidth of $B = 1.6$GB/s, CPU clock frequency of $f = 3$GHz, average memory access latency of $T_m = 250$ cycles, cache block size of $K = 64$ bytes, and the miss frequency of Thread 1 of $M_1A_1 = 2$ million/s. Note that these numbers are just one realistic and representative data point. Since we will only make qualitative observations, it is the trends that matter, rather than the actual numbers.

4.2.1 The Impact of Miss Frequency

Figure 4.1(a) plots the optimum vs. natural bandwidth share for Thread 2 when miss frequency of Thread 2 is varied against that of Thread 1, with prefetching turned on (prefOpt vs. prefNat, in dashed lines) or turned off (Opt vs. Nat, in solid lines). The prefetching coverage and accuracy are assumed to be $c_1 = 0.7$, $a_1 = 0.5$, $c_2 = 0.5$, and $a_2 = 0.2$.

Comparing Opt and Nat, they intersect at the point of equal miss frequencies ($\frac{M_2A_2}{M_1A_1} = 1$). At this intersection point, both of the natural bandwidth share already produces optimum weighted speedup. As we increase Thread 2's miss frequency by going to the right of the intersection along the x-axis, the natural and optimum bandwidth shares of Thread 2 increase, but the optimum bandwidth share increases at a slower pace. In essence, for optimum weighted speedup, the thread with higher miss frequency must be slightly (but not overly) constrained.

Comparing prefOpt and prefNat, they intersect at a point where the x-axis is roughly 0.5. At this point, the optimum bandwidth demand from misses and prefetches of both threads are equal. More specifically, according to Equation 4.20, when $\frac{M_{p,2}(M_{p,2}+P_2)A_2^2}{M_{p,1}(M_{p,1}+P_1)A_1^2} = 1$, $\frac{M_2A_2}{M_1A_1} = 0.583$. As we increase Thread 2's miss frequency by going to the right of the intersection along the x-axis, both of the natural and optimum bandwidth shares of Thread 2 increase. As before, the optimum bandwidth share increases at a slower pace. Thus we can make the following observation:

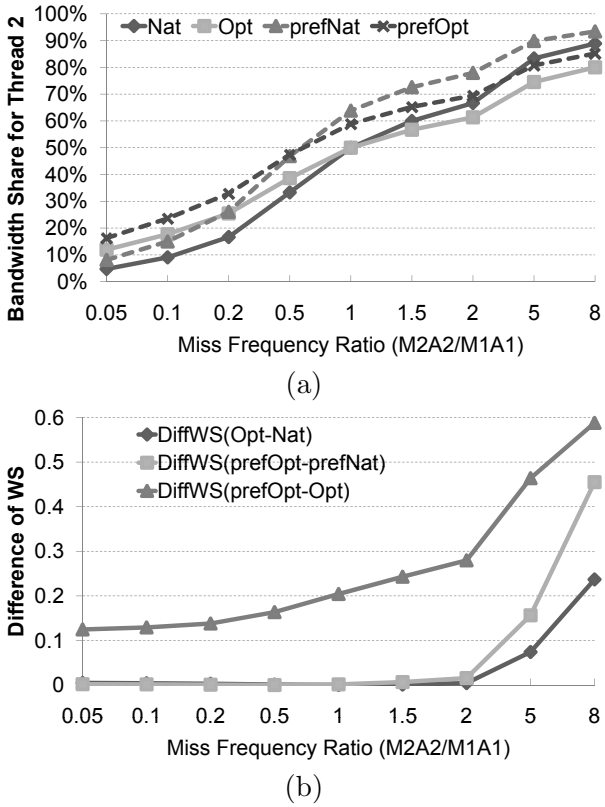


Figure 4.1: Bandwidth share for Thread 2 (a), and the resulting weighted speedup (b), as the miss frequency of Thread 2 varies.

Observation 12 *With or without prefetching, to produce optimum weighted speedup, the threads that have higher bandwidth demand (higher cache miss and prefetch frequencies) must be slightly (but not overly) constrained, in effect preventing bandwidth hungry applications from dominating the bandwidth usage and starving other applications.*

Comparing Opt and PrefOpt, PrefOpt is always above Opt, implying that Thread 2 always receives a larger bandwidth allocation when prefetching is applied, regardless of its relative miss frequency. Conversely, Thread 1's optimum bandwidth allocation is reduced after prefetching applied. Given that Thread 1 has higher prefetching coverage and accuracy than Thread 2, we can make the following observation that is opposite to conventional wisdom:

Observation 13 *Compared to a system without prefetching, the optimum bandwidth partitioning for a system with prefetching tends to constrain the bandwidth usage of a core that has relatively more useful prefetches (higher prefetching coverage and accuracy), in favor of slightly increasing the allocations for other cores with lower prefetching coverage and accuracy.*

The intuition behind this observation is that a thread having higher prefetching coverage and accuracy tends to utilize the off-chip bandwidth more efficiently; hence this thread can

donate some of its bandwidth allocation to other threads in order to improve the overall performance. We will further investigate how coverage alone (Section 4.2.2), as well as accuracy alone (Section 4.2.3), impacts the optimum bandwidth allocation and weighted speedup respectively, by assuming other parameters fixed.

How does the slight constraint on bandwidth allocation for the thread with higher bandwidth demand impact weighted speedup? Figure 4.1(b) shows the difference in weighted speedup between various scenarios. DiffWS(Opt-Nat) represents the improvement in weighted speedup that comes from bandwidth partitioning, when prefetching is not applied. The curve shows an increasing trend as we increase Thread 2's miss frequency along the x-axis. This is because Thread 2's higher miss frequency begins to saturate the off-chip bandwidth.

DiffWS(prefOpt-prefNat) represents the weighted speedup improvement that comes from bandwidth partitioning, when prefetching is applied. The curve shows the same trend as DiffWS(Opt-Nat), but with steeper increase when $\frac{M_2A_2}{M_1A_1} >= 1.5$. The steeper increase is caused by the additional bandwidth consumption from prefetching, causing two effects: scarcer bandwidth, and larger difference in bandwidth demand from the two threads. To illustrate the latter effect, we note that after prefetching, the bandwidth demand ratio of the two threads becomes $\frac{M_{p,2}(M_{p,2}+P_2)A_2^2}{M_{p,1}(M_{p,1}+P_1)A_1^2} = \frac{(1-c_2)(1-c_2+c_2/a_2)}{(1-c_1)(1-c_1+c_1/a_1)} \cdot \frac{M_2^2A_2^2}{M_1^2A_1^2} = 2.94\frac{M_2^2A_2^2}{M_1^2A_1^2}$, where $\frac{M_2^2A_2^2}{M_1^2A_1^2}$ is the bandwidth demand ratio before prefetching.

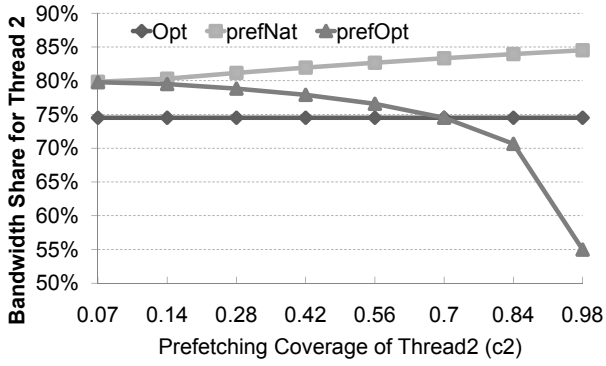
Notice that while prefOpt reallocates about 10% of the available bandwidth from Thread 2 to Thread 1 as compared to prefNat (Figure 4.1(a)), it produces significant improvement in weighted speedup when the off-chip bandwidth is scarce. To summarize:

Observation 14 *Prefetching increases the magnitude of improvement in system performance from bandwidth partitioning, if it increases the difference in bandwidth demand from different cores and makes bandwidth scarcer. In contrast, the magnitude of improvement may decrease if prefetching reduces the difference of bandwidth demand from different cores.*

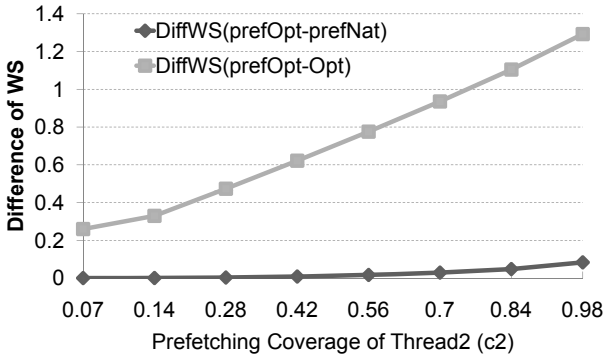
Analyzing DiffWS(prefOpt-Opt), we can conclude that deploying both prefetching and optimum bandwidth partitioning, the system can achieve better performance than only applying optimum bandwidth partitioning. An example of this effect can be seen in the co-schedule *hmmer-milc* (Figure 1.4(a)).

4.2.2 The Impact of Prefetching Coverage

Figure 4.2(a) shows Thread 2's natural and optimum bandwidth shares (prefNat and prefOpt) as its prefetching coverage varies along the x-axis. As for other parameters, Thread 1's coverage is set at 70% ($c_1 = 0.7$), while Thread 2's coverage varies from 7% to 98%. The prefetching accuracy for both threads is 70% ($a_1 = a_2 = 0.7$). The miss frequency ratio of two threads without prefetching is $\frac{M_2A_2}{M_1A_1} = 5$. This implies Thread 2's optimum bandwidth share without prefetching is 74.5% (the horizontal line labeled "Opt").



(a)



(b)

Figure 4.2: Bandwidth share for Thread 2 (a), and the resulting weighted speedup (b), as the prefetching coverage for Thread 2 varies.

As prefetching coverage of Thread 2 increases along the x-axis, the off-chip bandwidth demand from Thread 2 increases, since an accuracy of 70% means that for every cache miss eliminated, 1.4 prefetch requests are generated. The increase in bandwidth demand increases Thread 2's natural bandwidth share (prefNat), but notably decreases Thread 2's optimum bandwidth allocation (prefOpt). Such result seems to counter conventional wisdom, which may dictate that due to equal accuracy, prefetch requests from Thread 1 and Thread 2 are of equal importance. Consequently, higher prefetching coverage for Thread 2 should increase Thread 2's bandwidth allocation.

Our analytical model points out a flaw in the conventional wisdom. The reason why Thread 2's optimum bandwidth share decreases as its coverage increases is because Thread 2's performance improvement due to fewer cache misses comes at disproportionate decrease in Thread 1's performance, while Thread 1 has higher sensitivity to queueing delay for its memory requests. Thus, one cannot just consider prefetching coverage and accuracy of both threads, but must also consider the sensitivity of a thread's performance to the queuing delay.

Comparing Opt and prefOpt, they intersect at a point where Thread 1 and Thread 2 have

an equal prefetching coverage ($c_1 = c_2 = 0.7$), indicating that at equal prefetching coverage and accuracy, prefetching does not affect the optimum bandwidth partition sizes. As Thread 2's coverage increases further, its optimum bandwidth share decreases way below that before prefetching. The following observation summarizes the finding:

Observation 15 *The higher the prefetching coverage of a core, its bandwidth allocation must be decreased by a necessary amount in order to prevent it from dominating the bandwidth usage and starving other cores.*

How much does the optimum bandwidth partitioning improve weighted speedup? In Figure 4.2(b), DiffWS(prefOpt-prefNat)shows the weighted speedup improvement from bandwidth partitioning on a system with prefetching. The curve increases as Thread 2's coverage increases, due to both increasing scarcity of bandwidth, and increasing difference in bandwidth demand between the two threads, as concluded in Observation 14.

DiffWS(prefOpt-Opt) represents the weighted speedup improvement from prefetching at optimum bandwidth settings. The curve increases as Thread 2's coverage increases, because higher coverage implies more useful prefetching. Notice that DiffWS(prefOpt-Opt) is always above DiffWS(prefOpt-prefNat), because with relatively high prefetching accuracy ($a_1 = a_2 = 0.7$), prefetching makes larger impact on weighted speedup than optimum bandwidth partitioning, especially as Thread 2's coverage increases.

4.2.3 The Impact of Prefetching Accuracy

Figure 4.3(a) shows Thread 2's natural and optimum bandwidth share (prefNat and prefOpt) as its prefetching accuracy increases from 5% to 100%. Thread 1's accuracy is set at 10% ($a_1 = 0.1$). The prefetching coverage for both threads is 10% ($c_1 = c_2 = 0.1$). The miss frequency ratio of the two threads without prefetching is $\frac{M_2 A_2}{M_1 A_1} = 5$, implying that Thread 2's optimum bandwidth share without prefetching is 74.5% ("Opt").

Analyzing prefNat and prefOpt in the figure, prefOpt is always below prefNat, indicating that Thread 2, which has higher miss and prefetch frequencies, is slightly constrained in order to achieve optimum weighted speedup (instantiating Observation 12).

Conventional wisdom may dictate that to produce maximal performance, a thread that has higher prefetching accuracy should be rewarded with a larger share of bandwidth, so that its prefetch requests are prioritized over prefetch requests from other threads with lower accuracy. Figure 4.3(a) shows that conventional wisdom is only partially correct and is overall incorrect. As Thread 2's prefetching accuracy increases along the x-axis, fewer prefetch requests are needed to eliminate the same number of cache misses. This causes a decline in bandwidth usage, decreasing the natural bandwidth share (prefNat) used by Thread 2. The optimum bandwidth share (prefOpt) also declines, but at a slower pace. This gap between prefNat

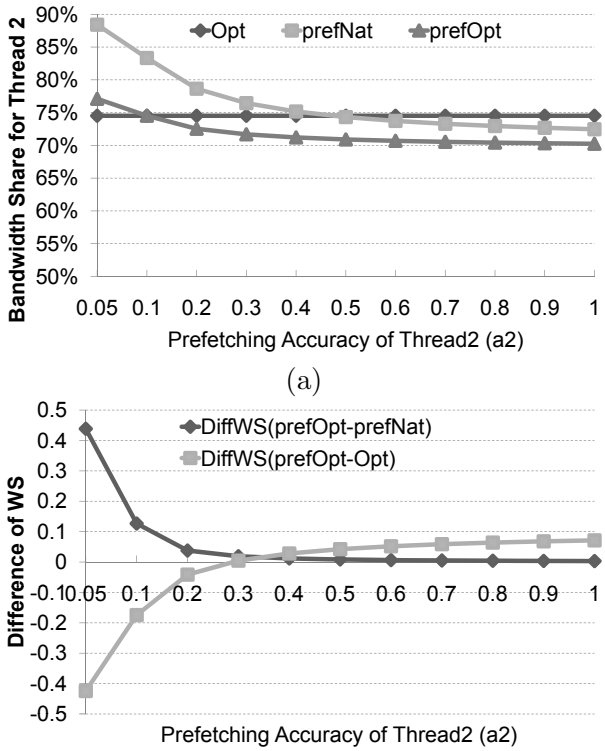


Figure 4.3: Bandwidth share for Thread 2 (a), and the resulting weighted speedup (b), as Thread 2's prefetching accuracy varies.

and prefOpt narrows as Thread 2's accuracy increases, indicating that Thread 2's optimum bandwidth allocation is less constrained with higher accuracy. While such narrowing supports conventional wisdom that more accurate prefetches should be given a higher priority because they use bandwidth more efficiently in eliminating cache misses, conventional wisdom is correct only in a relative sense. In an absolute sense, the bandwidth share allocated should decrease as prefetching accuracy increases.

Let us look at the optimum bandwidth partitioning after prefetching deployed (prefOpt) vs. without prefetching (Opt). They intersects at the point where Thread 2 and Thread 1's accuracy matches ($a_2 = a_1 = 0.1$). As Thread 2's accuracy increases along the x-axis, its optimum bandwidth share after prefetching falls below that before prefetching, while the opposite is true as we decrease Thread 2's accuracy. This indicates that when prefetching is highly accurate, the optimum bandwidth share should decrease. Thus:

Observation 16 *In an absolute term, the higher the prefetching accuracy for a core, the less bandwidth share should be allocated to it. In a relative term, more accurate prefetches should be given a relatively higher priority in bandwidth usage over less accurate prefetches.*

Figure 4.3(b) shows the impact of accuracy on weighted speedup. DiffWS(prefOpt-prefNat) represents the weighted speedup improvement due to bandwidth partitioning in a system with prefetching. The curve shows large improvement at low accuracy, but declines and flatlines with higher accuracy. This makes sense, because at high accuracy, bandwidth demand declines, making bandwidth less scarce. In addition, the difference of bandwidth demand between the two threads decreases ($\frac{(M_{p,2}+P_2)A_2}{(M_{p,1}+P_1)A_1}$ declines from 6.05 to 1.05). Both factors make bandwidth partitioning less effective (Observation 14). In contrast, DiffWS(prefOpt-Opt) increases along the higher prefetching accuracy. To summarize:

Observation 17 *As prefetching accuracy improves, prefetching becomes more effective, while bandwidth partitioning becomes less effective in improving system performance.*

Note that DiffWS(prefOpt-Opt) stays negative until Thread 2's accuracy exceeds 30%. This means that at low coverage and accuracy, prefetching may degrade performance so much that bandwidth partitioning cannot fully repair it. Only when prefetching is useful enough and the difference in miss and prefetch frequencies is high enough, can bandwidth partitioning fully repair the performance loss from prefetching. Thus:

Observation 18 *The decision to turn on or off prefetching, should be made prior to bandwidth partitioning, because bandwidth partitioning may not fully repair the performance loss due to low coverage and accuracy prefetching.*

This observation explains the reason for experimental results in Figure 1.4. Therefore, our prefetching profitability metric should be used to decide whether to selectively turn on/off prefetching for each core prior to making bandwidth partitioning decisions.

4.3 Empirical Evaluation

4.3.1 Environment and Methodology

Simulation Parameters. We use a cycle-accurate full system simulator based on Simics [60] to model a CMP system with dual and quad cores. Each core has a scalar in-order issue pipeline with 4GHz clock frequency. To remove the effect of cache contention between cores, each core has private L1 and L2 caches. The L1 instruction cache and data caches are 16KB, 2-way associative, and have a 2-cycle access latency. The L2 cache is 512KB, 8-way associative, and has an 8-cycle access latency. All caches use a 64-byte block size, implement write-back policy, and LRU replacement. The bus to off-chip memory is a split transaction bus, with a peak bandwidth of 800MB/s for a single core system, 1.6GB/s for dual-core CMP and 3.2GB/s for quad-core CMP. The average main memory access latency is 300 cycles. The simulator allocates contiguous physical memory to each application. The CMP runs Linux OS that comes with Fedora Core 4 distribution.

Hardware Prefetchers. The L2 cache for a core is installed with a typical stream prefetcher, similar to the hardware prefetchers used in previous studies [5, 45, 50, 69]. The stream buffers are commonly implemented in modern processors [7, 24, 29, 32] due to their low hardware cost and high effectiveness for a wide range of applications. Stream buffers can detect accesses to block addresses that form a sequential or stride pattern (called a *stream*), and prefetch the next few blocks in the stream in anticipation that the processor will continue to access blocks from that stream. We implement a stream prefetcher with four streams, and up to four blocks prefetched for each stream.

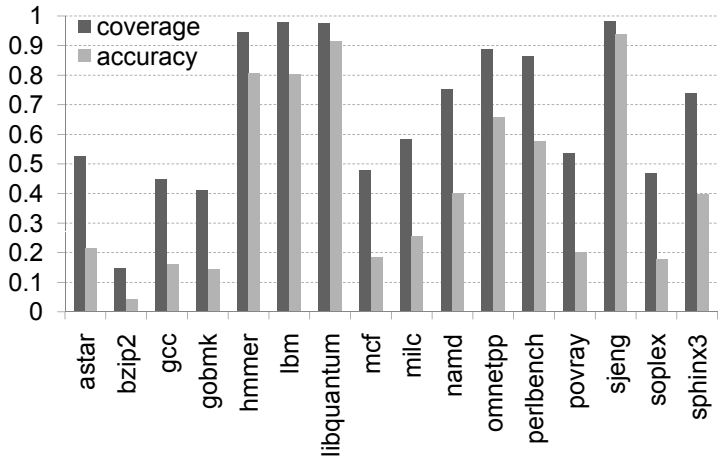


Figure 4.4: Benchmarks prefetch coverage and accuracy.

Workload Construction. We consider seventeen C/C++ benchmarks from the SPEC2006 benchmark suite [82]. We compile the benchmarks using `gcc` compiler with `O1` optimization level into `x86` binaries. We use the *ref* input sets, simulate 250 million instructions after skipping the initialization phase of each benchmark, by inserting breakpoints when major data structures have been initialized. We pair the benchmarks into co-schedules of two or four benchmarks. To reduce the number of co-schedules that we need to evaluate, we categorize the benchmarks based on the prefetching characteristics. Figure 4.4 shows the prefetch coverage and accuracy of each benchmark. From the figure, we pick up three representative benchmarks that have low (*bzip2*), medium (*astar*) and high (*hmmer*) coverage and accuracy, and pair each one with the other benchmarks to cover all representative co-schedules.

4.3.2 Experimental Results

Validating the Prefetching Metric

In this section, we investigate how well the composite prefetching metric in Equation 4.13 predicts prefetching performance profitability. We run each application on a single core system with available bandwidth of 800MB/s, and measure CPIs when prefetching is turned on (CPI_p) and when prefetching is turned off (CPI). Then we compute the difference ($\Delta CPI = CPI_p - CPI$) for each benchmark. A negative number represents performance improvement while a positive value represents degradation. Table 4.2 shows the applications sorted by their ΔCPIs. As a comparison, we compute our composite prefetching metric $\theta = \frac{MAK^2}{B^2} f(c-2+\frac{1-c}{a})$ and sort the applications in decreasing order of θ. Finally, conventional wisdom evaluates prefetching performance based on coverage or accuracy, so we also show the applications sorted in increasing order of accuracy. Note that Figure 4.4 shows that coverage and accuracy are highly correlated, hence we only show accuracy in the table. Applications whose ranks match with ones obtained from ΔCPI sorting are shown in bold.

Table 4.2: Ranking of applications based on $\Delta CPI = CPI_p - CPI$, our composite metric $\theta = \frac{MAK^2}{B^2} f(c-2+\frac{1-c}{a})$, and conventional metric accuracy a. The available bandwidth is 800MB/s.

ΔCPI-sorted			θ-sorted			a-sorted	
Benchmark	ΔCPI	Rank	Benchmark	θ	Rank	Benchmark	Rank
bzip2	4.837	1	**bzip2**	1375.77	1	**bzip2**	1
soplex	4.484	2	**soplex**	148.61	2	gobmk	4
mcf	4.463	3	**mcf**	145.88	3	gcc	5
gobmk	0.218	4	**gobmk**	85.44	4	soplex	2
gcc	0.207	5	**gcc**	78.25	5	mcf	3
povray	-0.006	6	astar	56.40	9	**povray**	6
namd	-0.012	7	milc	10.40	10	astar	9
omnetpp	-0.047	8	povray	0.83	6	milc	10
astar	-0.101	9	namd	-1.47	7	sphinx3	13
milc	-0.285	10	omnetpp	-7.92	8	namd	7
perlbench	-0.293	11	**perlbench**	-25.29	11	**perlbench**	11
hmmer	-1.364	12	**hmmer**	-43.11	12	omnetpp	8
sphinx3	-1.565	13	libquantum	-62.90	14	lbm	15
libquantum	-3.770	14	lbm	-69.39	15	hmmer	12
lbm	-4.196	15	sphinx3	-74.13	13	libquantum	14
sjeng	-13.820	16	**sjeng**	-76.86	16	**sjeng**	16

Let us compare the ranks of applications based on the actual performance improvement due to prefetching (ΔCPI), vs. based on our metric θ. The ranks of eight applications (shown in bold) out of sixteen exactly match. For the other eight applications, their ranks differ by at

most three positions, indicating how well our metric θ correlates with the actual performance. If each application is given a rank number, and we compare ranks based on ΔCPI vs. θ, the correlation coefficient computes to 95%, indicating high correlation between them. In contrast, the correlation of ranks based on ΔCPI vs. accuracy a computes to 89%. From the rank comparison, it is clear that our composite metric correlates much better with the actual performance than conventional metrics such as accuracy or coverage.

In addition, conventional metrics cannot determine what level of accuracy or coverage is high enough to produce performance improvement, whereas our composite metric θ includes a performance profitability threshold (Equation 4.13): $T_m > \frac{\theta}{\alpha}$, where $\alpha \in (0, 1)$ is a number that reflects how bursty memory accesses are. Even when α is unknown, θ can still determine prefetching profitability unambiguously in some cases. For example, $\theta = 1375.77$ for *bzip2*, which is much larger than the fully exposed memory access latency $T_m = 300$ cycles, hence θ predicts unambiguously that prefetching will hurt performance (Observation 9). In addition, there are eight applications showing negative θ values in Table 4.2. θ predicts unambiguously that prefetching will improve performance. Only for seven out of sixteen cases, the actual value of α is needed to determine whether prefetching is profitable. Now we will show how α can be estimated.

Table 4.3: Ranking of applications based on $\Delta CPI = CPI_p - CPI$, our composite metric $\theta = \frac{MAK^2}{B^2} f(c-2+\frac{1-c}{a})$, and conventional metric accuracy a. The available bandwidth is 1.6GB/s.

ΔCPI-sorted			θ-sorted			a-sorted	
Benchmark	ΔCPI	Rank	Benchmark	θ	Rank	Benchmark	Rank
bzip2	0.472	1	**bzip2**	469.57	1	**bzip2**	1
povray	-0.005	2	soplex	57.28	9	gobmk	5
namd	-0.012	3	mcf	56.95	10	gcc	6
omnetpp	-0.042	4	gobmk	24.13	5	soplex	9
gobmk	-0.052	5	gcc	21.68	6	mcf	10
gcc	-0.059	6	astar	20.71	7	povray	2
astar	-0.352	7	milc	3.36	8	**astar**	7
milc	-0.370	8	povray	0.21	2	**milc**	8
soplex	-0.478	9	namd	-0.37	3	sphinx3	14
mcf	-0.484	10	omnetpp	-2.04	4	namd	3
perlbench	-0.514	11	**perlbench**	-6.98	11	**perlbench**	11
hmmer	-1.040	12	**hmmer**	-13.20	12	omnetpp	4
libquantum	-2.636	13	**libquantum**	-22.23	13	lbm	15
sphinx3	-2.981	14	**lbm**	-25.02	14	hmmer	12
lbm	-3.244	15	**sphinx3**	-25.64	15	libquantum	13
sjeng	-10.433	16	**sjeng**	-30.07	16	**sjeng**	16

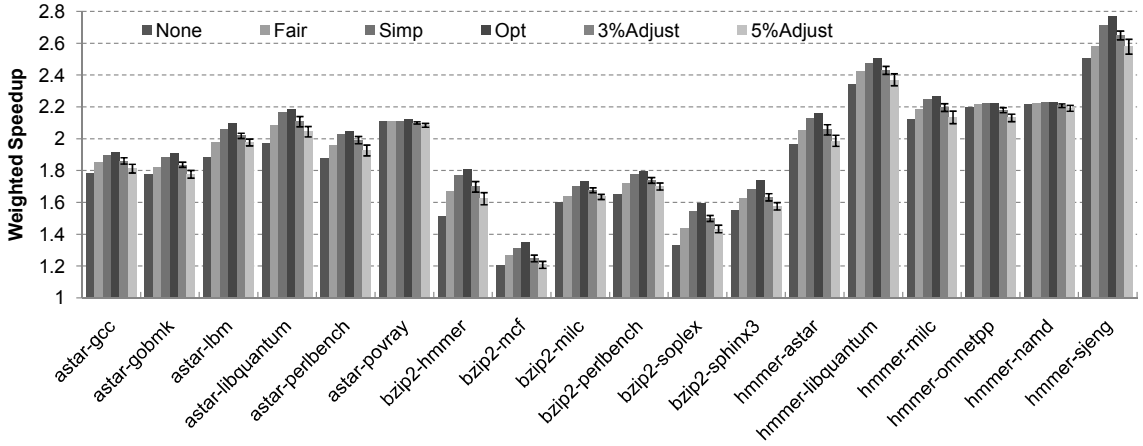
If we choose $\alpha = 0.2$, all benchmarks ranked above *astar* satisfy $T_m \leq \frac{\theta}{\alpha}$, while all benchmarks ranked at *astar* and below satisfy $T_m > \frac{\theta}{\alpha}$. Thus, 0.2 provides a reasonable estimate for α. To validate the goodness of the estimate $\alpha = 0.2$, we double the available bandwidth to $B = 1.6$GB/s, and re-run all the benchmarks to identify which benchmarks can benefit from prefetching. As shown in Table 4.3, due to the increase in available bandwidth, now only *bzip2* suffers from performance degradation, while all other benchmarks enjoy performance improvement. Using $\alpha = 0.2$, θ correctly predicts this outcome: only for *bzip2*, $\theta = 469.57 > T_m$, while for all other benchmarks $\frac{\theta}{0.2} < T_m$. The correlation coefficient of ranks between ΔCPI and θ is 69%, while it computes to 65% for ΔCPI and accuracy a. For further validation, we also tried another algorithm by controlling the prefetch aggressiveness, which directly changes the coverage and accuracy for all benchmarks (Table 4.4). The correlation coefficients of ranks based on ΔCPI and θ vs. based on ΔCPI and accuracy a are 95% and 75% resepectively. Again, $\alpha = 0.2$ predicts prefetching performance very well and hence can be used to decide whether to turn on/off prefetching.

Table 4.4: Ranking of applications based on $\Delta CPI = CPI_p - CPI$, our composite metric $\theta = \frac{MAK^2}{B^2} f(c - 2 + \frac{1-c}{a})$, and conventional metric accuracy a. The available bandwidth is 800MB/s.

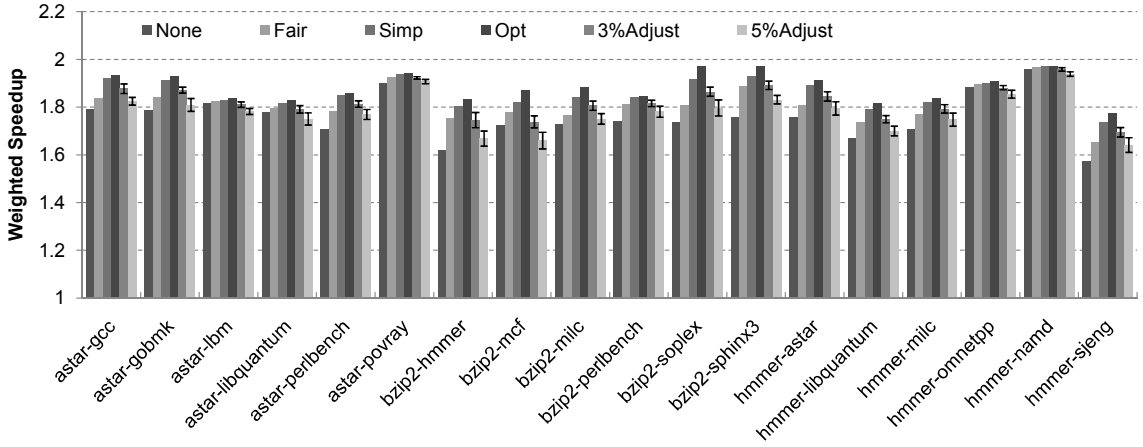
ΔCPI-sorted			θ-sorted			a-sorted		
Benchmark	ΔCPI	Rank	Benchmark	θ	Rank	Benchmark	a	Rank
povray	4.837	1	milc	1375.77	4	milc	0.362	4
namd	4.484	2	povray	148.61	1	povray	0.385	1
omnetpp	4.463	3	namd	145.88	2	gcc	0.466	5
milc	0.218	4	omnetpp	85.44	3	namd	0.529	2
gcc	0.207	5	**gcc**	78.25	5	astar	0.555	10
gobmk	-0.006	6	**gobmk**	56.40	6	**gobmk**	0.587	6
perlbench	-0.012	7	**perlbench**	10.40	7	bzip2	0.601	8
bzip2	-0.047	8	**bzip2**	0.83	8	sphinx3	0.635	12
hmmer	-0.101	9	**hmmer**	-1.47	9	soplex	0.644	11
astar	-0.285	10	**astar**	-7.92	10	perlbench	0.810	7
soplex	-0.293	11	**soplex**	-25.29	11	omnetpp	0.863	3
sphinx3	-1.364	12	**sphinx3**	-43.11	12	mcf	0.882	15
libquantum	-1.565	13	**libquantum**	-62.90	13	hmmer	0.890	9
lbm	-3.770	14	**lbm**	-69.39	14	libquantum	0.949	13
mcf	-4.196	15	**mcf**	-74.13	15	sjeng	0.973	16
sjeng	-13.820	16	**sjeng**	-76.86	16	lbm	0.983	14

Approximating Optimum Bandwidth Partition

Recall that Equation 4.20 provides the weighted speedup optimum bandwidth partitions for different cores with prefetchers turned on. Unfortunately, the equation cannot be implemented in hardware because the inputs $CPI_{alone,i}$ and $CPI_{L2\infty,i}$ cannot be easily obtained. To make a practical implementation, we assume $C_i = \frac{CPI_{alone,i}}{CPI_{L2\infty,i}} = 1$ and use it in the simplified partitioning scheme. The assumption no longer guarantees the optimality of performance, but we will demonstrate that the deviation from optimum is minor.



(a) Prefetchers of all cores are turned on



(b) Prefetchers of all cores are turned off

Figure 4.5: Weighted speedup for various partition schemes in a dual-core CMP system with prefetchers turned on (a) and prefetchers turned off (b).

The bandwidth partitioning is enforced using token bucket algorithm [57], where a token generator distributes tokens to different per-core buckets at the rates proportional to the fraction

of bandwidth allocated to different cores. Each cache miss or prefetch request is allowed to go to the off-chip interface only when the core generating the request has matching tokens in its bucket. In order to adapt to the changes in program behavior over time, we implement a dynamic partitioning scheme, where at the end of each interval of 1-million clock cycles, the miss and prefetch frequencies of various cores from the interval that just ended, are used to compute the new bandwidth shares for the next interval.

We test our bandwidth partitioning algorithm by comparing the weighted speedup with several schemes as shown in Figure 4.5: fair partitioning (Fair), which allocates the off-chip bandwidth equally among cores [66], simplified optimum partitioning (Simp), which assumes $C_i = 1$, and the true optimum partitioning (Opt), which relies on multiple simulation passes to collect the actual values of $CPI_{alone,i}$ and $CPI_{L2\infty,i}$, and allocations adjusted by 3% and 5% (x%Adjust) from Simp. Specifically, for each adjustment amount, we select one application and bump its bandwidth allocation by x%, and correspondingly reduce the allocation of the other by x%. There are two choices for each x% adjustment, so we show the average weighted speedup as a bar and the range (max and min) as a candle in the figure. For all cases, prefetchers of all core are either turned on or completely turned off.

The figure shows that compared to applying prefetching without bandwidth partitioning (None), fair partitioning (Fair) improves weighted speedup, primarily because to an extent it avoids pathological cases where one core dominates the bandwidth usage. However, our simplified bandwidth partitioning (Simp) outperforms Fair in all cases, because it dynamically chooses optimal bandwidth allocations. On average, the improvement in weighted speedup from Simp over None is twice as high as that from Fair, and is very close to the optimum partitioning (Opt), implying that assuming $C_i = 1$ does not cause Simp to deviate much from Opt.

When the bandwidth allocations are adjusted by even a small amount (3%) from the simplified allocations, weighted speedup degrades (or at best remains constant), comparing to Simp. The degradation increases with 5% adjustment. This implies that the bandwidth allocations achieved by Simp is within 3% deviation from Opt. Finally, the 3% and 5% adjustments also show that bandwidth partitioning can hurt performance if bandwidth allocations are not computed correctly, because bandwidth fragmentation may occur, in which memory requests that should have been served on the idle bus have to wait for new tokens, while other cores have tokens that they cannot utilize.

Validating Observations in Partitioning Model

In Section 4.2 we showed case studies using the bandwidth partition model and obtained several observations. In this section, we will validate these observations through empirical evaluation. Figure 4.6 and Figure 4.7 show the bandwidth share and weighted speedup respectively, for various application co-schedules, running on a dual-core CMP with four different configurations:

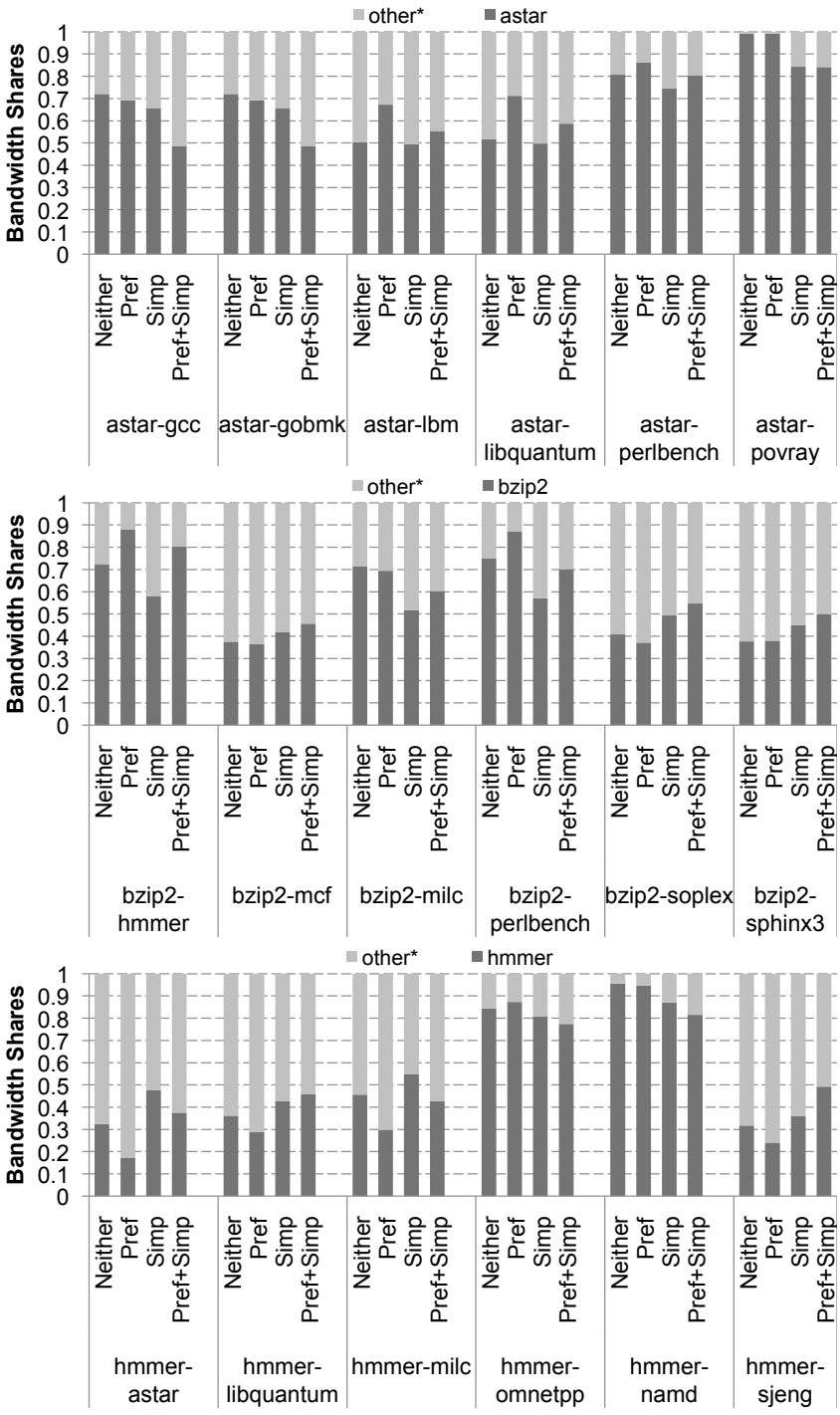


Figure 4.6: Bandwidth shares for various co-schedules under different configurations in a dual-core CMP.

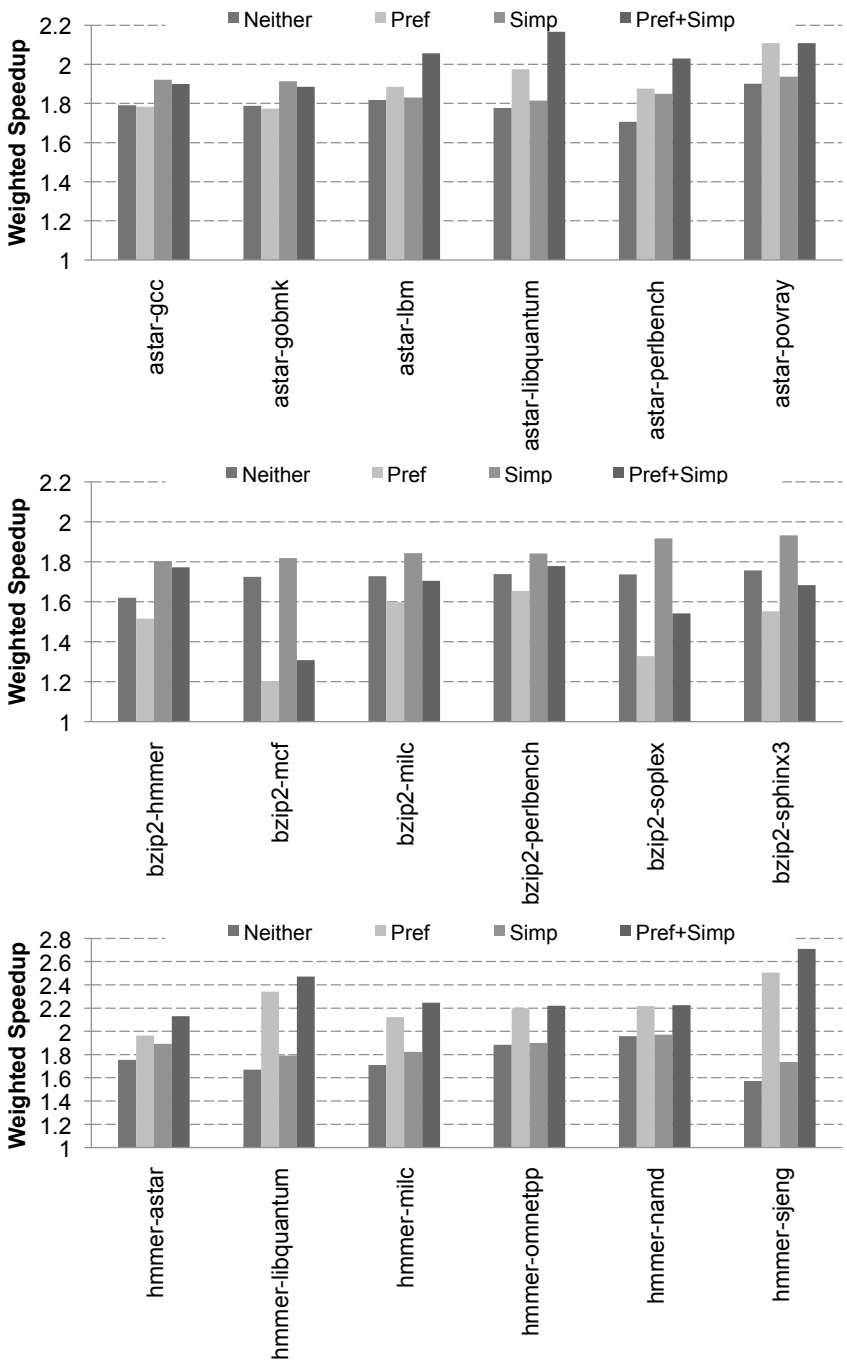


Figure 4.7: Weighted speedup for various co-schedules under different configurations in a dual-core CMP.

no prefetching or bandwidth partitioning (Neither), prefetching for both cores but no bandwidth partitioning (Pref), no prefetching but simplified bandwidth partitioning (Simp), as well as prefetching for both cores and simplified bandwidth partitioning (Pref+Simp). The available memory bandwidth is 1.6GB/s.

The bandwidth shares in Figure 4.6 show that when bandwidth partitioning is deployed, the application that has higher bandwidth demand is assigned smaller fraction of bandwidth, vs. when bandwidth partitioning is not used (Pref+Simp vs. Pref, and Simp vs. Neither). This agrees with Observation 12.

Comparing the bandwidth shares after prefetching vs. no prefetching (Pref+Simp vs. Pref) in Figure 4.6, the application with higher prefetching coverage and accuracy (Figure 4.4) receives smaller bandwidth share, validating Observation 13, Observation 15 and Observation 16.

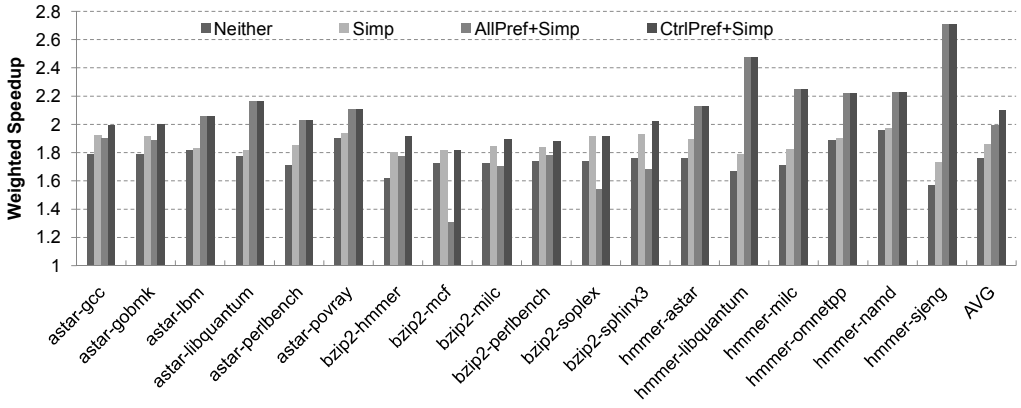
In Figure 4.7, we can see the cases (e.g. *astar-lbm* and *astar-libquantum*), Simp does not improve performance over Neither much. This is due to both applications in the co-schedules have roughly equal miss frequency (50%), hence their natural bandwidth shares already achieves optimum weighted speedup. However, after prefetching is applied, the weighted speedup improvement from bandwidth partitioning becomes large (Pref+Simp vs. Pref). The reason is because prefetching increases the difference in bandwidth demand of two cores, and therefore increases the impact of bandwidth partitioning on weighted speedup. This is also the case for most of the other co-schedules. In contrast, prefetching slightly decreases the difference in bandwidth demand (bandwidth shares Pref vs. Neither) in four co-schedules: *astar-gcc*, *astar-gobmk*, *bzip2-milc* and *bzip2-sphinx3*. This reduces the effectiveness of bandwidth partitioning, as can be seen from the smaller improvement of Pref+Simp over Pref, compared to the improvement of Simp over Neither. These results validate Observation 14.

For certain co-schedules such as *astar-povray*, *hmmer-omnetpp* and *hmmer-namd*, bandwidth partitioning does not affect performance much, regardless of prefetching (Simp vs. Neither, and Pref+Simp vs. Pref). The reason is that *povray*, *omnetpp* and *namd* are not bandwidth intensive due to the very few cache misses and prefetches. This makes the available bandwidth plentiful enough that bandwidth partitioning is not needed.

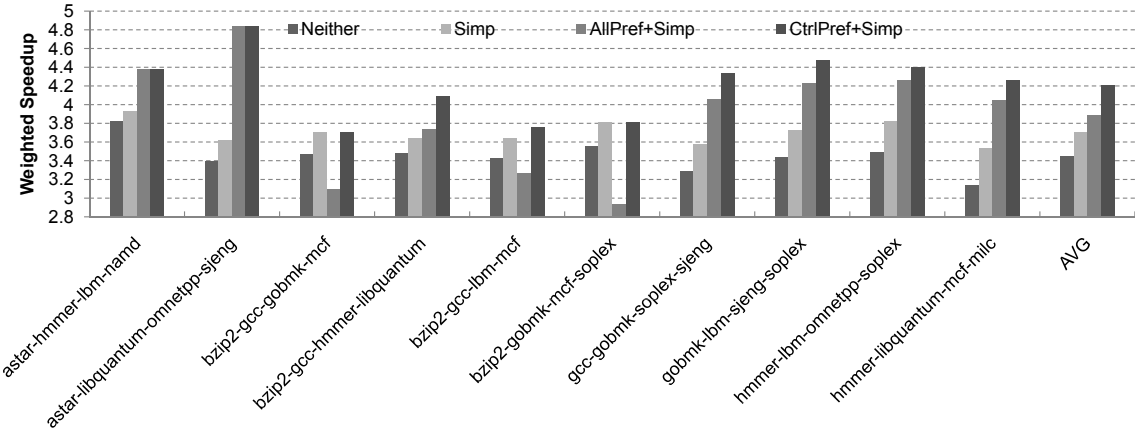
Finally, comparing the weighted speedup of Pref with Simp in Figure 4.7, we can see cases where prefetching improves weighted speedup more than bandwidth partitioning, such as in all co-schedules *hmmer* runs in. This is due to the high prefetching coverage and accuracy of *hmmer*, consistent with Observation 17.

Co-Deciding Prefetching and Partitioning

In this section, we show results where our prefetching profitability metric in Inequality 4.13 guides the decision to selectively turn on/off prefetchers, coupled with dynamic bandwidth partitioning.



(a) Weighted Speedup on a Dual-core CMP



(b) Weighted Speedup on a Quad-core CMP

Figure 4.8: Weighted speedup of optimum bandwidth partitioning for no prefetching, all prefetching, vs. selectively turning on prefetchers using the composite prefetching metric on dual-core CMP with 1.6GB/s bandwidth (a), and quad-core CMP with 3.2GB/s bandwidth (b).

Figure 4.8 shows the weighted speedup for four system configurations a CMP system: no prefetching or bandwidth partitioning (Neither), only simplified bandwidth partitioning (Simp), all prefetchers turned on and bandwidth partitioning (AllPref+Simp), and selective activation of each core's prefetcher guided by prefetching profitability metric coupled with bandwidth partitioning (CtrlPref+Simp). Specifically, we assume $\alpha = 0.2$ as discussed in Section 4.3.2. To make prefetching decision, we first run the applications on a CMP system with equal share of the available bandwidth for each core, and profile the hardware counters needed for computing θ. Based on the computed θ, we statically turn on/off the prefetcher of a core, and run the applications again to collect performance. Note that nothing prevents the prefetching decision to be performed dynamically at run time. However, making prefetching decision statically is sufficient to demonstrate the effectiveness of the prefetching profitability metric. The experiments

are evaluated on a dual-core CMP with available bandwidth of 1.6GB/s and on a quad-core CMP with available bandwidth of 3.2GB/s respectively.

The figure shows that in many cases, turning on all prefetchers (AllPref+Simp) degrades performance over no prefetching (Simp) (i.e. 8 out of 16 co-schedules on a dual core CMP, and in 3 out of 10 co-schedules on a quad-core CMP). Worse, performance of AllPref+Simp is even lower than the base case CMP without prefetching or bandwidth partitioning (Neither) in a majority of the above cases. This indicates that prefetching can be too harmful for bandwidth partitioning to offset, consistent with Observation 18.

In both figures, when we use our prefetching profitability criterion to guide the decision to turn on/off each core's prefetcher, coupled with dynamic bandwidth partitioning, the best weighted speedup can be achieved for all co-schedules. CtrlPref+Simp has at least the same performance as the best of Simp and AllPref+Simp, and in many cases outperforms both of them significantly. On average, over Neither, Simp, and AllPref+Simp, CtrlPref+Simp improves weighted speedup by 6.0%, 13.6% and 19.5%, respectively on a dual-core CMP, and 7.2%, 12.5% and 21.8%, respectively on a quad-core CMP.

4.4 Related Work

Bandwidth partitioning. Researchers have proposed various bandwidth partitioning techniques to mitigate CMP memory bandwidth contention-related problems. The implication is to prioritize memory requests from different cores based on heuristics, in order to meet the objectives of Quality of Service (QoS) [66], fairness [64, 65, 72], or throughput [33]. Liu et al. [57] investigated the impact of bandwidth partitioning on CMP system performance and its interaction with shared cache partitioning. Srikantaiah and Kandemir [80] explored a symbiotic partitioning scheme on the shared cache and off-chip bandwidth via empirical models. None of those bandwidth partitioning techniques took prefetching into account.

Prefetching. Hardware prefetching is widely implemented in commercial microprocessors [7, 24, 29, 32] to hide the long memory access latency. Due to the imperfect accuracy, prefetching incurs cache pollution and increases off-chip bandwidth consumption. Prior studies have looked into how to make prefetching more effective in CMP. Techniques include throttling/filtering useless prefetching requests [50, 51, 79, 81, 90] to reduce their extra bandwidth consumption, and a better hybrid prefetching algorithm [19, 67]. All of the above studies ignore bandwidth contention that arises from demand and prefetch requests coming from different cores.

Prefetching and Bandwidth Partitioning. Only recently, prefetching and bandwidth partitioning in CMP were studied together as inter-related problems. Ebrahimi et al. [18] proposed to partition bandwidth usage in the presence of prefetching in CMP in order to reduce the inter-core interference. While their results show improved system performance, many questions are

left unanswered. For example, what fundamental factors affect the effectiveness of prefetching and bandwidth partitioning? How do they interact with each other? Does combining prefetching and bandwidth partitioning always achieve better performance than using only one of them? What bandwidth shares can produce optimum system performance? The goal of this paper is to find out these answers, that are critical for designing a good policy for optimizing system performance. Due to the lack of understanding on these issues, the study [18] made a critical error of always keeping the prefetching engines of all cores on regardless of the situations. Figure 1.4 shows that in some cases, it is better to turn the prefetchers completely off.

4.5 Conclusions

The goal of this chapter is to understand how hardware prefetching and memory bandwidth partitioning in CMP can affect system performance and interact with each other. We have presented an analytical model to achieve this goal. Firstly, we derived a composite prefetching metric that can help determine under which situations hardware prefetcher can improve system performance. Then we constructed a bandwidth partitioning model that can derive weighted speedup-optimum bandwidth allocations among different cores. We showed three case studies based on the partitioning model and arrived at several interesting observations. For instance, prefetching can increase the impact of bandwidth partitioning on system performance by increasing the difference of bandwidth demand from different cores and making the available bandwidth scarcer. When prefetchers are turned on for all cores, weighted speedup optimum partitioning tends to assign more bandwidth to the cores running applications with lower coverage and accuracy, compared to without using prefetching. Finally, we collected simulation results to validate the observations obtained from the analytical model, and show system performance improvement that can be achieved by co-deciding prefetching and bandwidth partitioning decisions using our prefetching metric and implementable dynamic bandwidth partitioning.

Chapter 5

Retrospection

In modern processor system design, memory hierarchy performance is a key component toward the goal of exponential performance improvement projected by Moore's law. This dissertation makes several contributions in modeling the performance of the last level cache and the off-chip memory bandwidth in the memory hierarchy. In this chapter, Section 5.1 summarizes the contributions in the previous three chapters, and Section 5.2 reflects what I have learned during the past four years and offers my opinions based on the research.

5.1 Summary

In this dissertation, we apply analytical modeling approach to investigate the contention-related performance issues due to temporal sharing as well as spatial sharing in the last level cache and the off-chip memory bandwidth.

Firstly, we motivated from the growing trend of multi-programmed environment created by the emerging workloads/applications, such as server consolidation and virtualization. To allow multiple processes/threads of execution to time-share a limited number of processors, context switching mechanism is provided in modern computer systems. While very useful, context switching also introduces high performance overheads, with the primary reason of cache perturbation effect. We developed an analytical model to understand how cache parameters, a thread's temporal reuse pattern, and the amount of cache perturbation impact the number of context switch cache misses an application suffers from. Studying the model, we find that applications with larger reuse distance or flatter stack distance profile are more vulnerable from context switch misses. Using the model, we can analyze how prefetching and different cache sizes can affect the worst case of context switch misses for an application. Based on the observations in the case study, we also provide suggestions for architects on how to reduce the context switch misses and how to keep the actual number of cache misses lower than the worst case.

Secondly, as CMPs have become the mainstream computing platform, to an extent it reduces

the side effects of context switching on system performance, we moved the research toward CMPs. In CMPs, the expensive resources, such as the last level cache and off-chip memory bandwidth, are commonly shared by multiple cores, resulting in severely resource contention problems, in which an individual thread performance, as well as the system throughput are highly dependent on co-scheduled applications and how the resources are partitioned among them. It is well known how cache partitioning can improve system performance by reallocating cache capacity to reduce the total number of cache misses in the system, while unfortunately how bandwidth partitioning can affect system performance and how it interact with cache partitioning are not well understood. Therefore, we proposed an analytical model to investigate those factors and provide deeper insights for CMP architecture design. In the model, we can theoretically derive the bandwidth allocation for each core to optimize system throughput expressed as weighted speedup. Another contribution is that we simplify the mathematical model and implement a practical bandwidth partitioning scheme using token bucket algorithm to approximate the optimum system performance.

Thirdly, we find in most high performance processors today, levels of caches are augmented with hardware prefetching in order to hide long memory access latency. Since prefetching accuracy can never achieve 100%, the latency hiding benefit incurs costs in extra off-chip bandwidth usage and in cache pollution. Although memory bandwidth partitioning may improve bandwidth usage, how it interacts with hardware prefetching and how to deploy them in synergy to maximize system performance are not well studied. Therefore, we provide an analytical model to study the impact of hardware prefetching and memory bandwidth partitioning on CMP system performance. The proposed model has a prefetching metric, that helps decide under which conditions the hardware prefetching can improve system performance; a bandwidth partitioning model can take prefetching effects into account and derive the weighted speed optimum bandwidth allocations for different cores. Through model-driven study on three cases, we obtain several interesting observations that can be valuable for CMP system design and optimization. The empirical evaluation results validate all observations in the study and show the maximum system performance can be achieved by incorporating the composite prefetching metric and dynamic memory bandwidth partitioning.

5.2 Reflections

In this section, I discuss the research projects I have done in the past four years with my personal opinions on benefits and drawbacks of the motivation and methodology, and discuss some possible future work.

Analytical modeling is highly useful in the early stage of processor design, because it provides a fast and efficient method to estimate performance and obtain design insights and trade-offs.

Through analytical modeling, the architecture design process can be significantly improved in terms of resource efficiency, since the predicted performance can be easily obtained by tuning the parameters in the model. However, analytical modeling suffers from two major drawbacks. One is that it requires expertise in mathematical modeling, such as abstracting the architectural problem into high level mathematical relations, resorting to reasonable assumptions to make the model simple but feasible to produce general and correct insights. The other is the model prediction accuracy cannot be 100%, which means that it can be useful for diagnosis purpose, but processor verification and validation process has to count on more detailed and accurate simulations.

The motivation of context switch cache misses is likely good, because context switching is a mature mechanism in both hardware and software, and cache is ubiquitous in modern computer systems. In addition, emerging parallel workloads, such as server consolidation, virtual monitors, and thread level parallelism applications, make context switching more inevitable in high performance computing. Although the shift to CMPs may reduce the frequency of context switching in the system, the extra performance produced by CMPs has enabled increasingly complex software [47]. Therefore, context switch misses still exist in CMPs when the number of executing threads is more than the number of available cores.

The analytical model we developed for studying the behavior context switch misses can predict the trend of context context switch misses well with respect to cache perturbation amount. However, at the beginning of the model validation process, we always started with very high prediction errors. But this does not mean it is not a good model and we should start over or give up. Re-examining the constructed model against the simulations is very important. The high error level may come from unreasonable assumptions, mistakes in model construction, or simulation environment including simulator parameters and benchmark behavior. Starting from analyzing the simulation results and benchmark behavior, it is a long procedure to move back and forth between fixing model construction and modifying simulations, in order to reduce the error as much as possible. Never giving up and keeping thinking are the keys to systematically reduce validation errors. For instance, while the predicted numbers can capture the trend of the simulated results well for most benchmarks, from a single data point's view, the error between predicted and simulated results may still be high for some benchmarks. If we did not analyze the cache access pattern across different cache sets, we would never figure out using single cache set stack distance profile can further reduce the prediction errors for those benchmarks, although the extra cost is to collect per cache set stack distance profile.

The case studies on prefetching and cache sizes using the developed model are interesting, since they are well-known architecture techniques for performance enhancement. We explained why prefetching tends to increase the number of context switch misses and how to alleviate the side effects of prefetching by reducing its aggressiveness. In addition, we show that although

the natural cache misses can be significantly reduced as the cache size is increased, at fixed time quantum, the worst-case (maximum cache perturbation) context switch misses begin to eclipse the natural misses at some point. The results argue that on a large cache size, simply increasing time quantum (to reduce the context switch frequency) is not always the correct solutions. As a result, in addition to a thread's priority, OS scheduler design needs to take into account the cache size in the system. To reduce task switching overhead, Linux uses a default time slice of 100ms for a medium priority thread (nice value of 0); in addition, the rule of thumb adopted by Linux is to choose a time slice as long as possible to reduce task switching overhead, while keeping good system response time [9]. However, the CPU architecture design tends to integrate larger cache size in the future, placing an interesting dilemma for OS scheduler: on a very large cache size, the choice of time slice must be either relatively small so that the cache is only partially perturbed (good response time but high task switching overhead) or very large so that cache can stay filled up and reuse blocks for a while before being perturbed (small task switching overhead but bad response time). One challenging future work is to build an intelligent OS scheduler with low complexity and cost, that can adaptively assign different time slices to the running threads according to the dynamic execution environment, for the purpose of minimizing context switch misses and task switching overhead, while maintaining good response time.

Now the high performance computing processor has shifted toward multi-core design, which may to a extent reduce the extent of time-sharing CPU resource, has introduced an even more severe resource contention on the last level cache and off-chip memory bandwidth. The motivation of this project is despite of the wealth of studies in the last level cache, how off-chip memory bandwidth partitioning affects CMP system performance, how it interacts with cache partitioning as well as hardware prefetching in CMPs are poorly understood. Since little literature can be found to answer those important questions, we conduct analytical model based studies to investigate those unclear factors and concentrate on gaining valuable insights to confirm or correct conventional wisdom for CMP processor design. We also propose a practical implementation to enforce the derived theoretically optimum bandwidth allocations. The simulation results show that it delivers near optimum weighted speedup and performs better than fair partitioning and other bandwidth partitioning sizes. Incorporating hardware prefetching, even better system performance can be achieved by selectively turning on or off prefetchers on different cores and dynamic bandwidth partitioning.

In the era of multi-core design, performance optimization is not the sole objective any more, since power consumption is becoming increasingly important and is changing the way high performance processors are architected [76]. Power and energy efficient CMPs, that feature dynamic voltage and frequency scaling (DVFS), have become more popular in today's chip market. Despite the fact that traditional commercial processor design only provide full-chip

DVFS, such as Intel's Core Duo processors [2], prior research studies have demonstrated that DVFS at core granularity can offer more benefits [28, 34, 46, 53]. In addition, the chip market tends to move toward fine-grained DVFS. For example, AMD's quad-core Opteron [17] processor provides per-core frequency scaling, although not voltage scaling. Keeping this in mind, we reflect on the memory bandwidth partitioning scheme implemented using token bucket algorithm. An intuitive question may come out: can we implement bandwidth partitioning in the system and save power simultaneously?

The impact of bandwidth partitioning is essentially throttling the bandwidth-hungry applications to prevent them from dominating the bandwidth usage. During bandwidth partitioning, the core that runs a bandwidth-hungry application is throttled at the off-chip memory level, while the on-chip core units, including CPU and caches, simply stall and wait for memory requests to be served. This means the on-chip core does nothing but waste power consumption. If we can take advantage of the opportunity to scale down the frequency of that core (running bandwidth-hungry application), the same throttling effects can be achieved, without hurting the overall system throughput. Therefore, an alternative way to implement bandwidth partitioning is to deploy core-level DVFS. Remember that Observation 2 states the natural bandwidth share of a core uses is equal to the ratio of its miss frequency to the sum of all miss frequencies from all cores. To achieve the same effects as the optimum bandwidth partitioning through DVFS at core granularity, the natural bandwidth share for a core after deploying DVFS should be equal to the derived optimum bandwidth allocation.

Take a two-core CMP as an example, assuming the miss frequency of core 1 is larger than the miss frequency of core 2, and the miss frequency ratio is R (i.e. $R = \frac{M_1 A_1}{M_2 A_2} > 1$), then the optimum bandwidth allocation for core 1 $\beta_{1,opt}$ can be calculated from Equation 3.19 as:

$$\beta_{1,opt} = \frac{(M_1 A_1)^{2/3}}{(M_1 A_1)^{2/3} + (M_2 A_2)^{2/3}} = \frac{1}{1 + (\frac{1}{R})^{2/3}} \tag{5.1}$$

To achieve the optimum bandwidth allocation, core 1's operating frequency should be scaled down to $f_{1,s}$ from the original frequency f_1 (i.e. $f_{1,s} < f_1$), resulting in its cache access frequency after DFS ($A_{1,s}$) being scaled down accordingly:

$$\frac{A_{1,s}}{A_1} = \frac{f_{1,s}}{f_1} \tag{5.2}$$

After applying DFS, the natural bandwidth share for core 1 ($\beta_{1,nat}$) must be equal to its optimum bandwidth allocation ($\beta_{1,opt}$), in order to achieve the maximum weighted speedup. Therefore, we have the following equation:

$$\beta_{1,nat} = \frac{M_1 A_{1,s}}{M_1 A_{1,s} + M_2 A_2} = \frac{M_1 A_{1,s}}{M_1 A_{1,s} + \frac{M_1 A_1}{R}} = \beta_{1,opt} \tag{5.3}$$

Simplifying the above equation to express $A_{1,s}$ in terms of other parameters (A_1, R and $\beta_{1,opt}$):

$$A_{1,s} = \frac{\beta_{1,opt}}{R(1-\beta_{1,opt})} A_1 \tag{5.4}$$

Substituting Equation 5.1 and Equation 5.2 into Equation 5.4, we can obtain the DFS frequency of core 1:

$$\frac{f_{1,s}}{f_1} = \frac{1}{R^{1/3}} \tag{5.5}$$

The above equation can tell the original frequency should be scaled down by how much in order to achieve the optimum system throughput. Since the dynamic power consumed by a core is calculated as $P = CV^2f$, theoretically speaking, deploying DFS to achieve bandwidth partitioning can reduce the dynamic power consumption at the same pace with the factor the frequency is scaled down. An interesting future work is to implement power-efficient bandwidth partitioning through DFS in CMP systems.

REFERENCES

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

[2] A. Naveh and E. Rotem and A. Mendelson and S. Gochman and R. Chabukswar and K. Krishnan and A. Kumar. Power and Thermal Management in Intel Core Duo Processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[3] A. Snavely and D.M. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proc. of 19th International Conference on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 234–224, 2000.

[4] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov. 1988.

[5] A.R. Alameldeen. Using Compression to Improve Chip Multiprocessor Performance. 2006.

[6] A.R. Alameldeen and D.A. Wood. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*, 2007.

[7] B. Sinharoy and R.N. Kalla and J.M. Tendler and R.J. Eickemeyer and J.B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[8] R. Bitirgen, E. Ipek, and J.F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proc. of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[9] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[10] C. Cascaval and D.A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proc. of 17th Annual International Conference on Supercomputing*, pages 150–159, 2003.

[11] C. Cascaval, L. DeRose, D.A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Proc. of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 365–379, 1999.

[12] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting the Impact of Inter-Thread Cache Contention on a Chip Multiprocessor Architecture. In *Proc. of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351. IEEE Computer Society, 2005.

[13] J. Chang and G.S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proc. of International Conference on Supercomputing*, pages 242–252, 2007.

[14] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[15] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[16] F. M. David, J.C. Carlyle, and R.H. Campbell. Context Switch Overheads for Linux on ARM Platforms. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[17] J. Dorsey, S. Searles, M. Ciraula, E. Fang, S. Johnson, N. Bujanos, R. Kumar, D. Wu, M Braganza, and S. Meyers. An Integrated Quad-Core Opteron Processor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2007.

[18] E. Ebrahimi, O. Mutlu, C.J. Lee, and Y.N. Patt. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *Proc. of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[19] E. Ebrahimi, O. Mutlu, and Y.N. Patt. Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proc. of the 15th International Symposium on High Performance Computer Architecture(HPCA)*, 2009.

[20] P.G. Emma. Understanding Some Simple Processor-Performance Limits. *IBM Journal of Research and Development*, 41(3), 1997.

[21] F. Liu and Y. Solihin. Understanding the Behavior and Implications of Context Switch Misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):21:1–28, 2010.

[22] R. Fromm and N. Treuhaft. Revisiting the Cache Interference Costs of Context Switches. *http://citeseer.ist.psu.edu/252861.html*, 1996.

[23] G. Edward Suh and S. Devadas and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of International Conference on Supercomputing*, pages 1–12, 2001.

[24] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (1Q), 2001.

[25] F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. of the ACM SIGMETRICS/Performance 2006 Joint International Conference on Measurement and Modeling of Computer System (SIGMETRICS)*, pages 228–239, June 2006.

[26] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Qualify of Service in Chip Multi-Processors. In *Proc. of the 40th Annual IEEE/ACM Synopsium on Microarchitecture (MICRO)*, 2007.

[27] A. Hartstein, V. Srinivasan, T.R. Puzak, and P.G. Emma. On the Nature of Cache Miss Behavior: Is It $\sqrt{2}$? In *The Journal of Instruction-Level Parallelism*, volume 10, 2008.

[28] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip Multiprocessors. In *Proc. of the 2007 International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[29] H.Q. Le and W.J. Starke and J.S. Fields and F.O. Connell and D.Q. Nguyen and B.J. Ronchetti and W.M Sauer and E.M. Schwarz and M.T. Waden. IBM Power6 Microarchitecture. *IBM Journal of Research and Development*, 51:639–662, 2007.

[30] L.R. Hsu, S.K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, 2006.

[31] I.B. Vapnyarskii. Numerical Methods of Solving Problems of the Mathematical Theory of Standardization. *USSR Computational Mathematics and Mathematical Physics*, 18(2):484–487, 1978.

[32] IBM. *IBM Power4 System Architecture White Paper*, 2002.

[33] E. Ipek, O. Mutlu, J.F. Martinez, and R. Caruana. Self-Optimizing Memory Controller: A Reinforcement Learning Approach. In *Proc.of the 35th International Symposium on Computer Architecture (ISCA)*, 2008.

[34] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[35] ITRS. International Technology Roadmap for Semiconductors: 2005 Edition, Assembly and packaging. In *http://www.itrs.net/Links/2005ITRS/AP2005.pdf*, 2005.

[36] R. Iyer. CQoS: a Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proc. of the 18th Annual International Conference on Supercomputing*, pages 257–266, 2004.

[37] R. Iyer, L. Zhao, F. Guo, Y. Solihin, S. Markineni, D. Newell, R. Illikkal, L. Hsu, and S. Reinhardt. QoS Policy and Architecture for Cache/Memory in CMP Platforms. In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 2007.

[38] J. Renau and B. Fraguela and J. Tuck and W. Liu, M. Prvulovic and L. Ceze and S. Sarangi and P. Sack and K. Strauss and P. Montesinos. SESC. *http://sesc.sourceforge.net*, 2005.

[39] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms. In *Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, January 2010.

[40] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Integrating DRAM Caches for CMP Server Platforms. *IEEE Micro Top Picks*, 31(1):99–108, 2011.

[41] X. Jiang, A. Mishra, L. Zhao, R. Iyer, Z. Fang, S. Srinivasan, S. Makineni, P. Brett, and C.R. Das. ACCESS: Smart Scheduling for Asymmetric Cache CMPs. In *Proc. of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, February 2011.

[42] X. Jiang and Y. Solihin. Architectural Framework for Supporting Operating System Survivability. In *Proc. of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, February 2011.

[43] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *Proc. of the 18th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, September 2009.

[44] J.J. Yi and D.J. Lilja and B. Calder and L.K. John and J.E. Smith. The Future of Simulation: A Field of Dreams. *Computer*, 39(11):22–29, 2006.

[45] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[46] P. Juang, Q. Wu, L.S. Peh, M. Martonosi, and D.W. Clark. Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors. In *Proc. of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[47] S. Kim, F. Liu, Y. Solihin, R. Iyer, L. Zhao, and W. Cohen. Accelerating Full-System Simulation through Characterizing Predicting Operating System Performance. In *Proc. of IEEE International Symposium on Performance Analysis of System and Software (ISPASS)*, October 2008.

[48] P. Koka and M.H. Lipasti. Opportunities for Cache Friendly Process Scheduling. In *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[49] H. Kwak, B. Lee, A.R. Hurson, S. Yoon, and W. Hahn. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*, 48(2):176–184, Feb. 1999.

[50] C.J. Lee, O. Mutlu, V. Narasiman, and Y.N. Patt. Prefetch-Aware DRAM Controller. In *Proc. of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[51] C.J. Lee, V. Narasiman, O. Mutlu, and Y.N. Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *Proc. of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[52] C. Li, C. Ding, and K. Shen. Quantifying The Cost of Context Switch. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[53] J. Li and J.F. Martinez. Dynamic Power-Performance Adaption of Parallel Computation on Chip Multiprocessor. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[54] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real System. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture(HPCA)*, 2008.

[55] J.D.C. Little. A Proof of Queueing Formula L = λW. *Operations Research*, 9(383–387), 1961.

[56] F. Liu, F. Guo, S. Kim, A. Eker, and Y. Solihin. Characterizing and Modeling the Behavior of Context Switch Misses. In *Proc. of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, October 2008.

[57] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, January 2010.

[58] F. Liu and Y. Solihin. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip Multiprocessors. In *Proc. of ACM SIGMETRICS 2011 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2011.

[59] Y. Luo, O.M. Lubeck, H. Wasserman, F. Bassetti, and K.W. Cameron. Development and Validation of a Hierarchical Memory Model Incorporating CPU- and Memory-Operation Overlap Model. In *Proc. of the 1st International Workshop on Software and Performance*, pages 152–163, 1998.

[60] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer Society*, 35(2):50–58, 2002.

[61] R. L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, Nov 1970.

[62] Micron. 1Gb DDR2 SDRAM Component: MT47H128M8HQ-25. May 2007.

[63] J.C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Apr. 1991.

[64] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[65] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proc.of the 35th International Symposium on Computer Architecture (ISCA)*, 2008.

[66] K.J. Nesbit, D. Aggarwal, J. Laudon, and J.E. Smith. Fair Queuing Memory System. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[67] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[68] K.J Nesbit, J.Laudon, and J.E. Smith. Virtual Private Caches. In *Proc.of the 34th International Symposium on Computer Architecture (ISCA)*, pages 57–68, 2007.

[69] S. Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proc. of the 21st International Symposium on Computer Architecture*, pages 24–33, May 1994.

[70] M.K. Qureshi and Y.N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

[71] N. Rafique, W. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, pages 2–12, 2006.

[72] N. Rafique, W. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 2007.

[73] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory Access Scheduling. In *Proc.of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

[74] B. Rogers, A. Krishna, G. Bell, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proc. of the 36th International Conference on Computer Architecture (ISCA)*, 2009.

[75] E. Schnarr and J.R. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proc. of 8th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[76] J.S. Seng, D.M. Tullsen, and J.Z.N. Cai. Power Sensitive Multithreaded Architecture. In *Proc. of the 2000 Intertional Conference on Computer Design (ICCD)*, 2000.

[77] S.Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning on a Chip Multiprocessor Architecture. In *Proc.of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[78] Y. Solihin, V. Lam, and J. Torrellas. Scal-tool: Pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Supercomputing (SC)*, Nov 1999.

[79] L. Spracklen, Y. Chou, and S.G. Spracklen. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *Proc. of the 11th International Symposium on High Performance Computer Architecture(HPCA)*, 2004.

[80] S. Srikantaiah and M. Kandemir. SRP: Symbiotic Resource Partitioning of the Memory Hierarchy in CMPs. In *Proc. of International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[81] S. Srinath, O. Mutlu, H. Kim, and Y.N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-efficiency of Hardware Prefetchers. In *Proc. of the 13th International Symposium on High Performance Computer Architecture(HPCA)*, 2007.

[82] Standard Performance Evaluation Corporation. Spec cpu2006 benchmarks. *http://www.spec.org*, 2006.

[83] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of International Symposium on High Performance Computer Architecture*, pages 117–126, 2002.

[84] G.E. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(7–26), 2004.

[85] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.

[86] D. Thiebaut and H. S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[87] D. Tsafrir. The Context-Switching Overhead Inflicted by Handling Hardware Interrupts. In *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007.

[88] W.W. Hwu and T. Conte. The Susceptibility of Programs to Context Switching. *IEEE Transactions on Computers*, 43(9):994–1003, Sep. 1994.

[89] L. Zhao, R.R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring Large-Scale CMP Architectures Using ManySim. *IEEE Micro*, 27(4):21–31, 2007.

[90] X. Zhuang and H.-H.S. Lee. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. *IEEE Transactions on Computers*, 56(1):18–31, 2007.