ABSTRACT

GUO, FEI. Analyzing and Managing Shared Cache in Chip Multi-Processors. (Under the direction of Professor Yan Solihin).

Recently, Chip Multi-Processor (CMP) or multicore design has become the mainstream architecture choice for major microprocessor makers. In a CMP architecture, some important on-chip platform resources, such as the lowest level on-chip cache and the off-chip bandwidth, are shared by all the processor cores. As will be shown in this dissertation, resource sharing may lead to sub-optimal throughput, cache thrashing, thread starvation and priority inversion for the applications that fail to occupy sufficient resources to make good progress. In addition, resource sharing may also lead to a large *performance variation* for an individual application as determined by other applications that are co-scheduled with it. Such performance variation is ill-suited for the future uses of CMPs considering the recent trends in enterprise IT toward running a diverse set of applications that have diverse computing requirements. In this environment, many applications may require a certain level of performance guarantee, which we refer to as *performance Quality of Service* (QoS). A large performance variation is a major hindrance to provide QoS. As the number of cores in a CMP increases, the degree of sharing of platform resources can be expected to increase and will further exacerbate the performance variation problem. In this dissertation, we address the resource sharing problem from two aspects.

Firstly, in order to better understand what factors affect the degree of an application suffering from the impact of cache sharing, we propose an analytical and several heuristic models that encapsulate and predict the impact of cache sharing. The models differ by their complexity and prediction accuracy. We validate the models against a cycle-accurate simulation. The most accurate model achieves an average error of 3.9%. Through a case study, we found that the cache sharing impact is largely affected by the temporal reuse behaviors of the co-scheduled applications.

Secondly, in order to overcome the performance variation problem and provide deterministic throughput to the individual applications, we investigate a framework for providing performance Quality of Service in a CMP server. We found that the ability of a CMP to partition platform resources alone is not sufficient for fully providing QoS. We also need an appropriate way to specify a QoS target, and an admission control policy that accepts jobs only when their QoS targets can be satisfied. We also found that providing strict QoS often leads to a significant reduction in throughput due to resource fragmentation. We propose novel throughput optimization techniques that include: (1) exploiting various QoS execution modes, and (2) microarchitecture techniques that steal excess resources from a job while still meeting its QoS target. Through simulation, we found that compared to an unoptimized scheme, the throughput can be improved by up to 45%, making the throughput significantly closer to a non-QoS CMP.

Analyzing and Managing Shared Cache in Chip Multi-Processors

by
Fei Guo

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

2008

APPROVED BY:

Dr. Eric Rotenberg	Dr. Edward Gehringer
Dr. Yan Solihin Chair of Advisory Committee	Dr. Gregory Byrd

DEDICATION

To my beloved wife, Yifan, to my beautiful daughters, Tracy and Joyce, and to my parents Ruitang Guo and Junmin Cao.

BIOGRAPHY

FEI GUO was born in Wuhan, P. R. China in 1978. He was admitted by HuaZhong University of Science and Technology (HUST) and joined the distinguished Advanced Class in 1996. He obtained his B.S. degree in 2000, and his M.S. degree in 2002 from HUST. From 2002 to 2003, he was a software engineer in Guang Dong Science and Technology Academe of China Telecommunication. He has been a Ph.D. candidate in the Department of Electrical and Computer Engineering at North Carolina State University since spring 2004. He was an intern at Intel Corporation, Hillsboro, OR, in the summer 2006.

His research interests include supporting Quality of Service in chip multi-processors and performance modeling. He has authored/co-authored several conference papers and journal articles, and has released an Analytical CAche Performance Prediction (ACAPP) tool suite for public distribution.

ACKNOWLEDGMENTS

First, I wish to express my gratitude to my research advisor, Dr. Yan Solihin. I could not have completed this dissertation without his invaluable guidance and support. In addition, he has always encouraged me to challenge myself to be a successful researcher, which has helped me to fully discover my potential.

I would also like to offer my sincere thanks to my advisory committee members, Dr. Gregory Byrd, Dr. Edward Gehringer, and Dr. Eric Rotenberg, for their important feedback and suggestions with regard to this dissertation. I also highly appreciate the suggestions given by Dr. Frank Mueller, who was initially a member of my advisory committee.

I am thankful to my current and former fellow graduate students in the ARPERS research group for their kind help and friendship: Dhruba Chandra, Mazen Kharbutli, Seongbeom Kim, Brian Rogers, Xiaowei Jiang, Fang Liu, Abhik Sarkar, and Siddhartha Chhabra. My heartfelt appreciation especially goes out to Dhruba Chandra for his contribution to this dissertation and for helping me start my first research project.

I would additionally like to thank my colleagues at Intel Corporation: Donald Newell, Ravishankar Iyer, Li Zhao, Rameshi Illikkal, and Srihari Makineni. They provided me with a great working environment and valuable comments to help me start this dissertation. I would like to specially thank Li Zhao and Ravishankar Iyer for their contributions to this dissertation and for the collaboration on various research projects. In addition, my thanks go to Hari Kannan, an intern from Stanford University, for working closely with me on a project at Intel.

Finally, I would like to take this opportunity to express my deep gratitude to my wife, Yifan Zhou. She has sacrificed a lot to support and encourage me throughout my study. I am so lucky to be the recipient of her love. I am also very grateful to my parents, Ruitang Guo and Jummin Cao. They have devoted themselves to raising and supporting me. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Recently, Chip Multi-Processor (CMP) or multicore design has become the mainstream architecture choice for major microprocessor makers. Compared to single core design, CMPs provide throughput improvement for multi-threaded and multi-programmed workloads. However, since some important on-chip platform resources, such as the lowest level on-chip cache and the off-chip bandwidth, are shared by all the processor cores, the performance of co-scheduled applications running on a CMP highly depend on their ability to compete for the shared resources. As will be demonstrated in this dissertation, resource sharing affects applications non-uniformly. Some applications may be slowed down significantly, while others may not be. This may lead to sub-optimal throughput, cache thrashing, thread starvation and priority inversion for the applications that fail to acquire sufficient resources to make good progress. In addition, resource sharing may also lead to a large *performance* *variation* for an individual application as determined by other applications that are co-scheduled with it. Such performance variation is ill-suited for the future uses of CMPs considering the recent trends in enterprise IT toward running a diverse set of applications that have diverse computing requirements [40]. In this environment, many applications may require a certain level of performance guarantee, which we refer to as *performance Quality of Service* (QoS). A large performance variation is a major hindrance to providing QoS. As the number of cores in a CMP increases, the degree of sharing of platform resources can be expected to increase and will further exacerbate the performance variation problem.

To illustrate the performance variation problem due to cache sharing, Figure 1.1 shows an example in which *mcf* is co-scheduled with another application that runs on a different processor core in a 2-core CMP with private L1 cache and shared L2 cache. Figure 1.1(a) shows the impact of cache sharing on the IPC of each application in a co-schedule. The full height of each bar (black+white sections) represents the IPC of the application when it runs alone [1]. The black section represents the IPC of the application when it is co-scheduled with another application. From Figure 1.1(a), we can see that the impact of cache sharing is not uniform. When *mcf* is co-scheduled with *mst*, neither *mcf* nor *mst* suffers from much performance degradation due to cache sharing. When *mcf* is co-scheduled with *art* and *swim*, only *mcf* suffers from a significant slowdown. On the contrary, when *mcf* is co-scheduled with *gzip*, only *gzip*

[1]In Figure 1.1(a), the number of instructions that *mcf* executes in each co-schedule varies when it is co-scheduled with different benchmarks, making the IPC of *mcf* different in each co-schedule. The full evaluation setup can be found in Section 2.3.

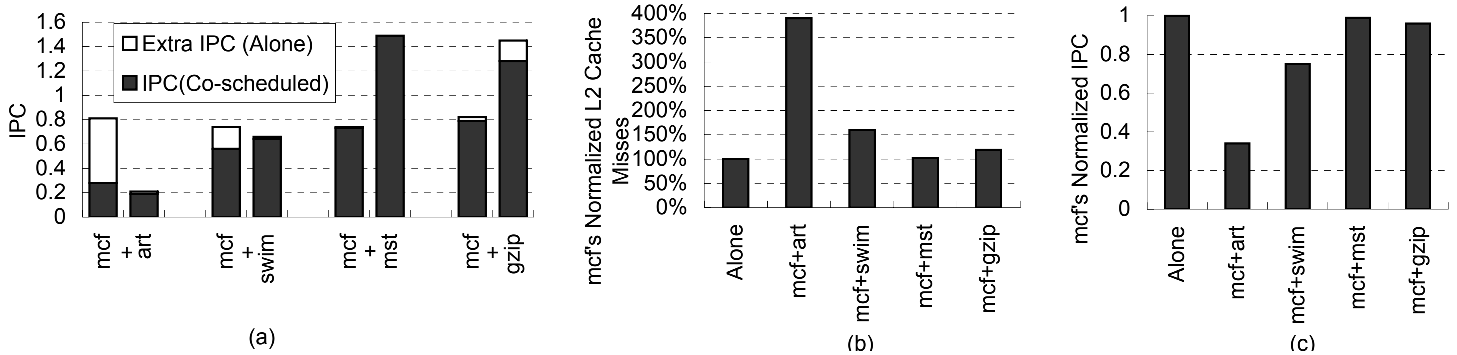


Figure 1.1: The IPCs of co-scheduled applications (a), the number of L2 cache misses (b), and IPC for *mcf* when it runs alone compared to when it is co-scheduled with another application (c). The L2 cache is 512KB, 8-way associative, and has a 64-byte line size. The full evaluation setup can be found in Section 2.3.

shows a considerable slowdown. Figure 1.1(b) and Figure 1.1(c) present the impact of cache sharing on *mcf* in terms of its L2 cache misses and its IPC with various co-scheduled applications, normalized to the case in which *mcf* runs alone.

In Figure 1.1(b) and Figure 1.1(c), *mcf* shows a very large performance variation across different co-schedules. When *mcf* runs together with *mst* or *gzip*, *mcf* does not suffer much from performance degradation. Its cache misses and IPC are still close to the case when it runs alone. However, when it runs together with *art* or *swim*, its number of L2 misses increases to roughly 390% and 160%, respectively, resulting in IPC reduction of 65% and 25%, respectively. From Figure 1.1, we can conclude that the impact of cache sharing is neither uniform nor consistent across different applications and across different co-schedules. In order to better understand what factors affect the degree of an application suffering from the impact of cache sharing, we propose an analytical and several heuristic models that encapsulate and predict the impact of cache sharing. In addition, in order to overcome the performance variation problem and provide deterministic throughput to the individual applications, we also propose a framework for providing performance Quality of Service. We will describe these two studies separately in the following sections.

1.1 Predicting Inter-Thread Cache Contention

The performance impact of cache sharing has been investigated by several researchers. Past studies have investigated profiling techniques that *detect* the cache sharing problems, and apply inter-thread cache partitioning schemes to improve fairness [25] or throughput [11, 25, 36, 48]. However, two important aspects were not addressed in past studies. The first aspect deals with determining what factors influence the cache sharing impact that a thread in a co-schedule suffers. The second aspect deals with whether such problems are predictable and hence preventable by knowing the factors. This dissertation addresses both questions by presenting an analytical and several heuristic models to predict the impact of cache sharing.

Past performance prediction models only predict the number of cache misses in a uniprocessor system [6, 9, 13, 14, 16, 26, 52, 55], or predict cache contention on a single processor time-shared system [1, 49, 50], where it was assumed that only one thread runs at any given time. Therefore, interference between threads that share a cache was not modeled.

This dissertation goes beyond past studies and presents three tractable models that predict the impact of cache space contention between threads that simultaneously share the L2 cache on a Chip Multi-Processor (CMP) architecture [7]. Two models, *Frequency of Access* (FOA) and *Stack Distance Competition* model (SDC), are based on heuristics. The third model is an analytical inductive probability model (*Prob*). The input to our models is the isolated L2 cache stack distance or *circular sequence* profiling of each thread, which can be easily obtained on-line or off-line. The output of the models is the number of extra L2 cache misses of each thread that shares the cache. We validate the models by comparing the predicted number of cache misses under cache sharing for fourteen pairs of benchmarks against a detailed, cycle-accurate CMP architecture simulation. We found that *Prob* is very accurate, achieving an average absolute error of only 3.9%. The two heuristics-based models are simpler but not as accurate.

Finally, the *Prob* model provides a valuable and practical tool through which we can study the impact of cache sharing extensively. We present a case study that evaluates how different temporal reuse behavior in applications influence the impact of cache sharing suffered by them. The case study gives an insight into what types of applications are vulnerable (or not vulnerable) to a large increase in cache misses under sharing.

1.2 Providing Performance Quality of Service

The primary goal of providing Quality of Service (QoS) is to guarantee a certain level of performance to individual applications. The need for such performance guarantee is motivated by the recent trends in the enterprise IT toward service-oriented computing, server consolidation, and virtualization. For example, in a service-oriented or utility computing environment, the utility computing provider may set up different *service-level agreements* (SLAs) to different clients that encapsulate guarantees in performance, reliability, manageability, and other metrics [34, 40]. Such an environment would require the server to be able to allocate platform resources proportionally to the level of the performance guarantee for each workload. For example, a job from a client with a “gold” SLA may be allocated more resources (e.g. larger cache sizes, higher processor count, and off-chip bandwidth) compared to jobs with standard SLAs. In virtualization, a virtual machine manager (VMM) hosts multiple virtual machines (VMs), where each VM runs a guest Operating System for various purposes ranging from regular to critical computations. It is beneficial if the VMM can allocate more platform resources to critical VMs and fewer resources to regular VMs. Finally, many transaction processing applications in a service-oriented computing domain would require a minimum level of real-time performance to be guaranteed. Overall, we believe that it is critical to support QoS in CMPs.

In recent studies, researchers have introduced frameworks in which applications specify their QoS targets, expressed in instructions per cycle (IPC) or resource performance (e.g. cache miss rates), while a resource manager dynamically partitions shared resources in order to meet each application’s QoS target [15, 20, 21, 22, 33, 37]. Unfortunately, these frameworks are insufficient if one wants to fully provide QoS in CMPs. Figure 1.2 shows an example in which multiple jobs (each job runs the SPEC2006 benchmark *bzip2*) run on a 4-core CMP with private L1 caches but a shared L2 cache. Let us assume that each job’s QoS target is to reach an IPC of at least 0.25, which is $\frac{2}{3}$ of its IPC when it runs alone. If the resource manager tries to satisfy the QoS targets of all jobs, it will equally divide the L2 cache among all instances of *bzip2*. However, from this figure, we can observe that while the jobs’ QoS targets are met when only two jobs run simultaneously, they are not met when three or four jobs run in the CMP. There are two major reasons why such frameworks fail in meeting the QoS targets of the jobs. First, the CMP does not have the ability to check whether its available resources are sufficient to satisfy a job's IPC target or the amount of resources needed by the job to meet its IPC target. Secondly, the lack of an admission control policy means that jobs are always accepted and run even when their QoS targets cannot be met.

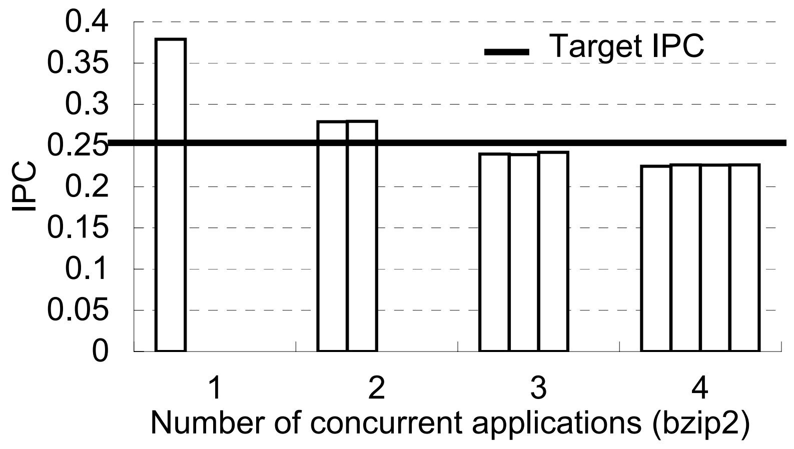


Figure 1.2: The IPC of different numbers of instances of *bzip2* running on a 4-core CMP with a 32-KB L1 cache per core, and a 2MB L2 cache shared by all cores. The full evaluation setup can be found in Section 3.5.

In this dissertation, we investigate a framework that is needed to fully provide QoS in a CMP system [17]. First, we study how a QoS target should be specified in order to enable the system to compare its available computation capacity against the requested computation capacity. We found that popular metrics (IPC and miss rate) are not suitable for this purpose, while capacity specification (e.g. cache size) naturally makes it easier to compare available computation capacity with demanded capacity. We then add an admission control policy to ensure that jobs are accepted only when their QoS targets can be satisfied.

Secondly, we introduce three *QoS execution modes*, as a way for jobs to specify how flexible they are with regard to their QoS targets. These execution modes are needed to match users' diverse requirements for their workloads as well as to provide a means for the system to boost throughput.

We use our framework to enable QoS in a CMP and apply it for managing processor cores and the shared L2 cache resources. We found that providing a strict QoS guarantee usually comes at a cost of significant reduction in system throughput due to jobs overspecifying their QoS targets, resulting in fragmentation of various resources in the system. We propose and investigate two techniques to recover the lost throughput. The first technique speculatively downgrades a job's QoS execution mode in order to boost the overall throughput. The second is a simple microarchitecture technique that we refer to as *resource stealing*, which steals cache capacity from a job which may have excess resources while still meeting the job's QoS target. We analyze the limitation of the original resource stealing technique and propose two improved resource stealing techniques to more effectively steal cache capacity. In addition, we propose a dynamic resource stealing mechanism to optimize the resource stealing for higher throughput.

We evaluate the proposed schemes and mechanisms on a 4-core CMP machine model based on Simics, with a recent version of the Linux Operating System, and workloads constructed by using SPEC2006 benchmarks. We found that through a combination of appropriate QoS target specification, admission control, and execution modes, a CMP can ensure that all accepted jobs have their QoS targets satisfied. In addition, we found that the proposed QoS execution modes and resource stealing mechanism are effective in improving throughput without violating jobs' QoS targets. They achieve throughput improvement between 13% and 45%, making the throughput significantly closer to a non-QoS CMP.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 describes the proposed models that predict the inter-thread cache contention in a CMP architecture. Chapter 3 presents the proposed framework that provides performance QoS in a CMP architecture. Finally, Chapter 4 concludes the dissertation.

Chapter 2

Predicting Inter-Thread Cache Contention in Chip Multi-Processors

This chapter is organized as follows. Section 2.1 presents the three models. Section 2.2 shows the hardware support for *Prob*'s circular sequence on-line profiling. Section 2.3 details the validation setup for our models. Section 2.4 presents and discusses the model validation results and the case study. The related works are discussed in Section 2.5. Finally, we conclude this study in Section 2.6.

2.1 Cache Miss Prediction Models

This section will present our three cache miss prediction models. It starts by presenting assumptions used by the models (Section 2.1.1), then it presents an overview of the three models (Section 2.1.2), the *frequency of access* (FOA) model (Section 2.1.3), the *stack distance competition* (SDC) model (Section 2.1.4), and the *inductive probability* (*Prob*) model (Section 2.1.5). Since the most accurate model is *Prob*, the discussion will focus mostly on *Prob*.

2.1.1 Assumptions

We assume that each thread's temporal behavior can be captured by a single stack distance or *circular sequence* profile. Although applications change their temporal behavior over time, in practice we find that the average behavior is good enough to produce an accurate prediction of the cache sharing impact. Representing an application with multiple profiles that represent different program phases may improve the prediction accuracy further, at the expense of extra complexity due to phase detection and profiling, e.g., [41]. This is beyond the scope of this dissertation.

It is also assumed that the profile of a thread is the same with or without sharing the cache with other threads. This assumption ignores the impact of the multi-level cache inclusion property [2]. In such a system, when a cache line is replaced from the L2 cache, the copy of the line in the L1 cache is invalidated. As a result, the L1 cache may suffer extra cache misses. This changes the cache miss stream of the L1 cache, potentially changing the profile at the L2 cache level. In the evaluation (Section 2.4), we relax the assumption and find negligible difference in the average prediction error (0.4%).

Co-scheduled threads are assumed not to share any address space. This is mostly true in the case where the co-scheduled threads are from different applications. Although parallel program threads may share a large amount of data, the threads are likely to have similar characteristics, making the cache sharing prediction easier because the cache is likely to be equally divided by the threads and each thread is likely to be impacted in the same way. Consequently, we ignore this case.

Furthermore, for most of the analyses, the L2 cache only stores data and not instructions. If instructions are stored in the L2 cache with data, the accuracy of the model decreases slightly (by 0.8%).

Finally, the L2 cache is assumed to use Least Recently Used (LRU) replacement policy. Although some implementations use different replacement policies, they are usually an approximation to LRU. Therefore, the observations of cache sharing impacts made in this dissertation are likely to be applicable to other implementations as well.

2.1.2 Model Overview

Stack Distance Profiling. The input to the models is the isolated L2 cache *stack distance* or *circular sequence* profile of each thread without cache sharing. A stack distance profile captures the temporal reuse behavior of an application in a fully- or set-associative cache [6, 26, 32], and is sometimes also referred to as marginal gain counters [49, 48]. For an A-way associative cache with LRU replacement algorithm, there are $A + 1$ counters: $C_1, C_2, \ldots, C_A, C_{>A}$. On each cache access, one of the counters is incremented. If it is a cache access to a line in the i^{th} position in the LRU stack of the set, C_i is incremented. Note that our first line in the stack is the most recently used line in the set, and the last line in the stack is the least recently used line in the set. If it is a cache miss, the line is not found in the LRU stack, resulting in incrementing the miss counter $C_{>A}$. A stack distance profile can be easily obtained statically by the compiler [6], by simulation, or by running the thread alone in the system [48].

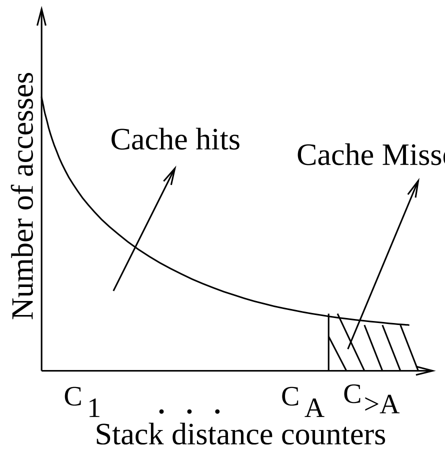


Figure 2.1: Illustration of a stack distance profile.

Figure 2.1 shows an example of a stack distance profile. Applications with temporal reuse behavior usually access more-recently-used data more frequently than less-recently-used data. Therefore, typically, the stack distance profile shows decreasing values as we go to the right, as shown in Figure 2.1. It is well known that the number of cache misses for a smaller cache can be easily computed using the stack distance profile. For example, for a smaller cache that has A' associativity, where $A' < A$, the new number of misses can be computed as:

$$miss = C_{>A} + \sum_{i=A'+1}^{A} C_i \tag{2.1}$$

For our purpose, since we need to compare stack distance profiles from different applications, it is useful to take the counter's frequency by dividing each of the counters by the number of processor cycles in which the profile is collected (i.e., $Cf_i = \frac{C_i}{CPUcycle}$). Furthermore, we refer to $Cf_{>A}$ as the *miss frequency*, denoting the frequency of cache misses in CPU cycles. We also refer to the sum of all other counters, i.e. $\sum_{i=1}^{A} Cf_i$ as *reuse frequency*. We refer to the sum of miss and reuse frequency as *access frequency* (Af).

Determining Factors. When several threads share a cache, they compete for cache space. Each thread ends up occupying a portion of the cache space, which we will refer to as the *effective cache space* of the thread. The size of the effective cache space determines the impact of cache sharing on the threads. A thread that succeeds in competing for sufficient cache space, relative to its working set, suffers less impact from cache sharing. The ability of a thread to compete for sufficient cache space, as will be discussed in Section 2.4.5, is determined by its temporal reuse behavior, which is largely determined by its stack distance profile. Intuitively, a thread that frequently brings in new cache lines (high miss frequency) and reuses them (high reuse frequency) has a higher effective cache space compared to other threads with low miss and reuse frequencies. Finally, although less obvious, a thread with a more concentrated stack distance profile (i.e., for all i, C_i is much larger than C_{i+1}) reuses fewer lines often, making the lines less likely to be replaced, increasing the effective cache space. Listing these factors helps to qualitatively distinguish how much detail each model takes into account.

Overview of the Models. We propose three models that vary in complexity and accuracy. Two of them are heuristics-based models: *frequency of access* (FOA) and *stack distance competition* (SDC). The two approaches are compared in Figure 2.2. Figure 2.2(a) illustrates how FOA and SDC predict the number of extra cache misses. Using a set of heuristics, they first predict the effective cache space (A') that a thread

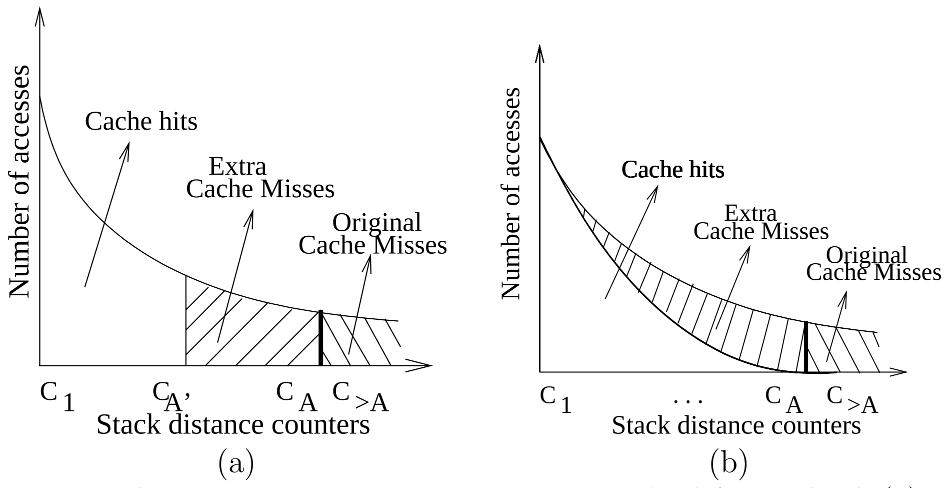


Figure 2.2: Comparing the prediction approach of FOA and SDC (a), with that of *Prob* (b).

will have under cache sharing. Then A' is input into Equation 2.1 to predict the number of misses that the thread will suffer under cache sharing (the shaded region in Figure 2.2(a)). This approach uses an assumption that accesses to more recently used lines, as long as their reuse distances are less than A', will not result in cache misses. However, this assumption may not be accurate in some cases. For example, even an access to the most recently used line may become a cache miss, if there are sufficient intervening accesses and misses from another thread. This aspect is taken into account by the *Prob* model, which computes the probability of each cache hit turning into a cache miss, for every possible access interleaving by an interfering thread. This is illustrated in Figure 2.2(b).

Table 2.1 compares the three models based on which of the three factors that determine the effective cache space of a thread under cache sharing are considered in the models. The table shows that FOA only takes into account the access frequency (sum of reuse and miss frequency). SDC takes into account both the reuse frequency and stack distance shape, but ignores the miss frequency. Finally, *Prob* takes into account all three factors but requires slightly more detailed profiling compared to the stack distance profiling, which will be discussed further in Section 2.1.5. From the table, we expect *Prob* to be the most accurate because it takes into account all the factors.

Table 2.1: Factors that determine the effective cache space a thread occupies.

Information Considered	FOA	SDC	Prob
Miss frequency	Partially	No	Yes
Reuse frequency	Partially	Yes	Yes
Stack dist profile shape	No	Yes	Yes
Profiling required	Stack distance	Stack distance	Circular sequence

2.1.3 Frequency of Access (FOA) Model

The simplest model, *frequency of access* (FOA), uses an assumption that the effective cache space of a thread is proportional to its access frequency. This assumption makes sense because a thread that has a high access frequency tends to bring in more data into the cache and retains them, occupying a larger effective cache space.

Let $CacheSize$ denote the total cache size, and Af_X denote the access frequency of thread X. Let us assume that there are N threads running together and sharing the cache. The effective cache size for thread X can be calculated from:

$$effCacheSize_X = \frac{Af_X}{\sum_{j=1}^{N} Af_j} \times CacheSize \tag{2.2}$$

By taking $A' = \frac{effCacheSize_X}{numCacheSet}$, and plugging it into Equation 2.1 and applying linear interpolation whenever necessary, we can obtain the number of cache misses under cache sharing. Since FOA only takes into account the access frequency, it may become inaccurate if the co-scheduled threads have different stack distance profile shapes, or when their ratios of miss and reuse frequency are very different.

2.1.4 Stack Distance Competition (SDC) Model

The *stack distance competition* (SDC) model tries to construct a new *combined* stack distance profile that merges individual stack distance profiles of threads that run together, by taking a subset of counters from each individual profile. This model relies on the intuition that a thread that reuses its lines the most will likely occupy more of the cache space than other threads. Based on such intuition, we consider the reuse frequency of each thread's stack distance position, starting from the MRU position as its current stack position being considered. For each (any) cache way in a set, the threads compete for the way, and the winner (the thread that is the likeliest to occupy the way) is the one with the highest reuse frequency at its current stack position. We then assign the way to the winner thread. The winner thread then advances its current stack position, and is ready for the next round of competition. The algorithm stops once all cache ways in the set are assigned to winner threads. We then count how many ways each thread has won, and compute its effective cache size based on that.

The algorithm shown in Figure 2.3 describes the algorithm steps in detail. Consider that we have n threads competing for the cache (denoted as $X_1, X_2, \ldots, X_n$) and the resulting combined behavior is denoted by Y. Let $C_i[X_j]$ denote the counter value of the i^{th} stack distance position of thread X_j, and $currPtr[X_j]$ denote the current stack position being considered for competition for thread X_j. Similarly, let $C_i[Y]$ denote the counter value of the i^{th} stack distance position of Y, and $currPtr[Y]$ denote the current stack position being competed for. Finally, let A denote the associativity of the cache.

Initially (Step 1), the *currPtr*'s for all threads and for Y are set to 1 so that they point to the first stack position in each stack distance profile. Then, in Step 2 the counters pointed by the current pointers from all threads are compared, and the thread with the maximum counter value is selected as the winner thread (say, X_{win}). Then, the current stack distance counter for the winner thread is copied into the combined stack distance profile, and the current pointers of both X_{win} and Y are incremented for the next competition. This step is repeated A times until the combined stack distance is fully populated.

```
SDC algorithm:
  1. Initialization
     For all thread $X_i$, $currPtr[X_i] = 1$; $currPtr[Y] = 1$;
  2. Repeat $A$ times:
     Find $X_{win}$ such that $currPtr[X_{win}] \geq currPtr[X_i], \forall i$
     $C_{currPtr[Y]}[Y] \leftarrow C_{currPtr[X_{win}]}[X_{win}]$
     $currPtr[Y] \leftarrow currPtr[Y] + 1$;
     $currPtr[X_{win}] \leftarrow currPtr[X_{win}] + 1$;
  3. Compute Cache Misses:
     For each thread $X_i$, its effective cache space is $\frac{currPtr[X_i]-1}{A} \times CacheSize$
```

Figure 2.3: SDC algorithm.

Figure 2.4 illustrates the end result of combining two stack distances of application

1 in Figure 2.4(a) and application 2 in Figure 2.4(b), into the combined stack distance profile in Figure 2.4(c), assuming an 8-way associative cache. At the end of the algorithm, the current pointer for application 1 is 4, and for application 2 is 6. The model predicts that application 1 will get $\frac{4-1}{8} = 37.5\%$ of the cache space, whereas application 2 will get $\frac{6-1}{8} = 62.5\%$ of the cache space.

The stack distance competition model is intuitive because it assumes that the higher the reuse frequency, the larger the effective cache space. However, it does not take into account the miss frequency. Therefore, the model can be inaccurate if the threads have very different miss frequency. It is also possible for the model to predict that a thread has a zero effective cache size, which is inaccurate.

2.1.5 Inductive Probability (*Prob*) Model

The most detailed model is an analytical model which uses inductive probability for predicting the cache sharing impact. Before we explain the model, it is useful to define two terms.

Definition 1 *A* **sequence** *of accesses from thread X, denoted as $seq_X(d_X, n_X)$, is a series of n_X cache accesses to d_X distinct line addresses by thread X, where all the accesses* map to the same cache set.

Definition 2 *A* **circular sequence** *of accesses from thread X, denoted as $cseq_X(d_X, n_X)$, is a special case of $seq_X(d_X, n_X)$ where the first and the last accesses are to the same line address, and there are no other accesses to that address.*

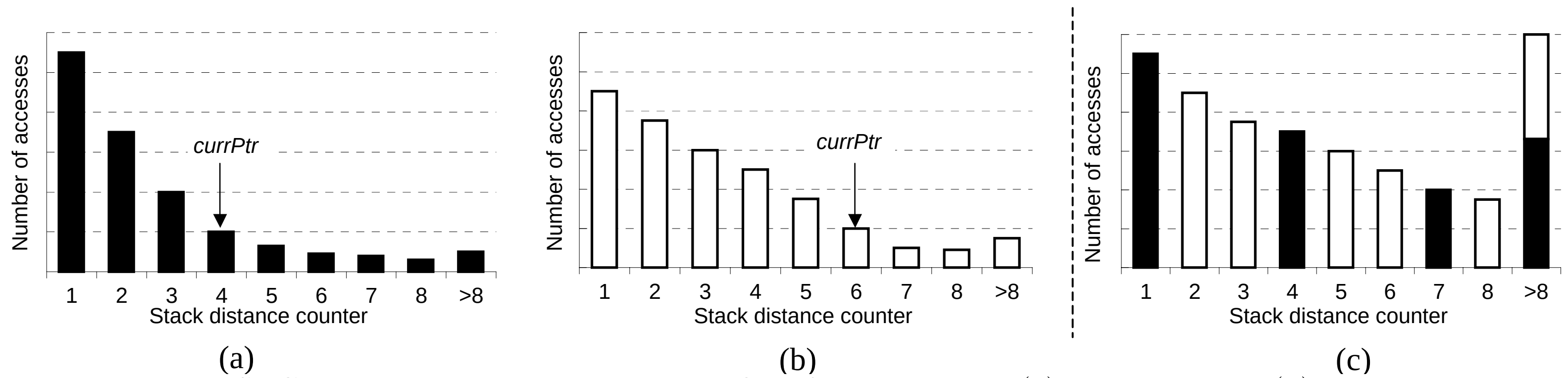


Figure 2.4: SDC illustration: stack distance profile of application 1 (a), application 2 (b), and the new combined stack distance (c). For the sake of completeness, although not required by the SDC model, the combined $Cf_{>A}$ is obtained by summing up all other bars that are not selected in the combined stack distance.

For a sequence $seq_X(d_X, n_X)$, $n_X \geq d_X$ necessarily holds. When $n_X = d_X$, each access is to a distinct address. We use $seq_X(d_X, *)$ to denote all sequences where $n_X \geq d_X$. For a circular sequence $cseq_X(d_X, n_X)$, $n_X \geq d_X + 1$ necessarily holds. When $n_X = d_X + 1$, each access is to a distinct address, except the first and the last accesses. We use $cseq_X(d_X, *)$ to denote all sequences where $n_X \geq d_X + 1$.

In a sequence, there may be several, possibly overlapping, circular sequences. The relationship of a sequence and circular sequences is illustrated in Figure 2.5. In the figure, there are eight accesses to five different line addresses that map to a cache set. In it, there are three circular sequences that are overlapping: one that starts and ends with address A ($cseq(4, 5)$), another one that starts and ends with address B ($cseq(5, 7)$), and another one that starts and ends with address E ($cseq(1, 2)$).

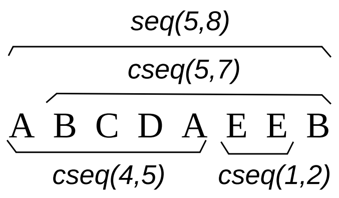


Figure 2.5: Illustration of the relationship between a sequence and circular sequences.

We are interested in determining *whether the last access of a circular sequence $cseq_X(d_X, n_X)$ is a cache hit or a cache miss* [1]. To achieve that, it is important to consider the following property.

[1]Note that, there are some addresses that are accessed only once. They do not form circular sequences. Each access results in a compulsory cache miss. Therefore, with or without cache sharing, the access remains a cache miss.

Property 1 *In an A-way associative LRU cache, the last access in a circular sequence $cseq_X(d_X, n_X)$ results in a cache miss if between the first and the last access, there are accesses to at least A distinct addresses (from any threads). Otherwise, the last access is a cache hit.*

Explanation: If there are accesses to a total of at least A distinct addresses between the first access up to the time right before the last access occurs, the address of the first and last access will have been shifted out of the LRU stack by the other A (or more) addresses, causing the last access to be a cache miss. If there is only $a < A$ distinct addresses between the first and the last access, then right before the last access, the address would be in the $(a+1)^{th}$ position in the LRU stack, resulting in a cache hit.

Corollary 1 *When a thread runs alone, the last access in a circular sequence $cseq_X(d_X, n_X)$ results in a cache miss if $d_X > A$, or a cache hit if $d_X \leq A$. Furthermore, in stack distance profiling, the last access of $cseq_X(d_X, n_X)$ results in an increment to the counter $C_{>A}$ if $d_X > A$ (a cache miss), or the counter C_{d_X} if $d_X \leq A$ (a cache hit).*

The corollary is intuitive since when a thread X runs alone, the number of distinct addresses in the circular sequence $cseq_X(d_X, n_X)$is d_X (because they only come from thread X). More importantly, however, the corollary shows the relationship between stack distance profiling and circular sequences. Every time a circular sequence with $d_X \leq A$ distinct addresses appears, C_{d_X} is incremented. If $N(cseq_X(d_X,*))$ denotes number of occurrences of circular sequences $cseq_X(d_X,*)$, we have $C_{d_X} = N(cseq_X(d_X,*))$. This leads to the following corollary.

Corollary 2 *The probability of occurrences of circular sequences $cseq_X(d_X,*)$ from thread X is equal to $P(cseq_X(d_X,*)) = \frac{C_{d_X}}{totAccess_X}$, where $totAccess_X$ denote the total number of accesses of thread X, and $d_X \leq A$.*

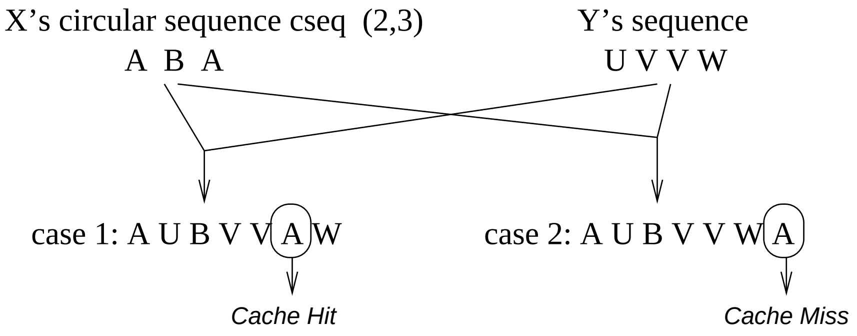


Figure 2.6: Illustration of how intervening accesses from another thread determines whether the last access of a circular sequence will be a cache hit or a miss. Capital letters in a sequence represent line addresses. The figure assumes a 4-way associative cache and all accesses are to a single cache set.

Let us now consider the impact of running a thread X together with another thread Y that shares the L2 cache with it. Figure 2.6 illustrates the impact, assuming a 4-way associative cache. It shows a circular sequence of thread X (" A B A"). When thread Y runs together and shares the cache, many access interleaving cases between accesses from thread X and Y are possible. The figure shows two of the access interleaving cases. In the first case, sequence "U V V" from thread Y occurs during the circular sequence. Since there are only three distinct addresses (U, B, and V) between the first and the last access to A, the last access to A is a cache hit. However, in the second case, sequence "U V V W" from thread Y occurs during the circular sequence. Therefore there are four distinct addresses (U, B, V, and W) between the accesses to A, which is equal to the cache associativity. By the time the second access to A occurs, address A is no longer in the LRU stack since it has been replaced from the cache, resulting in a cache miss for the last access to A. More formally, we can state the condition for a cache miss in the following corollary.

Corollary 3 *Suppose a thread X runs together with another thread Y. Also suppose that during the time interval between the first and the last access of X's circular sequence, denoted by $T(cseq_X(d_X, n_X))$, a sequence of addresses from thread Y (i.e., $seq_Y(d_Y, n_Y)$) occurs. The last access of X's circular sequence results in a cache miss if $d_X + d_Y > A$, or a cache hit if $d_X + d_Y \leq A$.* [2]

Every cache miss of thread X remains a cache miss under cache sharing. However, some of the cache hits of thread X may become cache misses under cache sharing, as implied by the corollary. The corollary implies that the probability of the last access in a circular sequence $cseq_X(d_X, n_X)$, where $d_X < A$, to become a cache miss is equal to the probability of the occurrence of sequences $seq_Y(d_Y, *)$ where $d_Y > A - d_X$.

Note that we now deal with a probability computation with four random variables (d_X, n_X, d_Y, and n_Y). To simplify the computation, we represent n_X and n_Y by their

[2]For simplicity, we only discuss a case where two threads share a cache. The corollary can easily be extended to the case where there are more than two threads.

expected values: $\overline{n_X}$ and $E(n_Y)$, respectively. Hence, Corollary 3 can be formally stated as:

$$P_{miss}(cseq_X(d_X, \overline{n_X})) = \sum_{d_Y = A - d_X + 1}^{E(n_Y)} P(seq_Y(d_Y, E(n_Y))) \tag{2.3}$$

Therefore, computing the extra cache misses suffered by thread X under cache sharing can be accomplished by using the following steps:

1. For each possible value of d_X, compute the weighted average of n_X (i.e. $\overline{n_X}$) by considering the distribution of $cseq_X(d_X, n_X)$. This requires a *circular sequence profiling*, which we will describe later. Then, we use $cseq_X(d_X, \overline{n_X})$ instead of $cseq_X(d_X, n_X)$.

2. Compute the expected time interval duration of the circular sequence of X, i.e. $T(cseq_X(d_X, \overline{n_X}))$.

3. Compute the expected number of accesses of Y, i.e. $E(n_Y)$, during time interval $T(cseq_X(d_X, \overline{n_X}))$. Then, use $seq_Y(d_Y, E(n_Y))$ to represent $seq_Y(d_Y, n_Y)$

4. For each possible value of d_Y, compute the probability of occurrence of the sequence $seq_Y(d_Y, E(n_Y))$, i.e. $P(seq_Y(d_Y, E(n_Y)))$. Then, compute the probability of the last access of X's circular sequence becoming a cache miss by using Equation 2.3.

5. Compute the expected extra number of cache misses by multiplying the probability of cache misses of each circular sequence with its number of occurrences.

6. Repeat Step 1-5 for each co-scheduled thread (e.g. thread Y).

We will now describe how each step is performed.

Step 1: Computing $\overline{n_X}$

$\overline{n_X}$ is computed by taking its average over all possible values of n_X:

$$\overline{n_X} = \frac{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X)) \times n_X}{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X))} \tag{2.4}$$

To obtain $N(cseq_X(d_X, n_X))$, an off-line profiling or simulation can be used. An on-line profiling is also possible, using simple hardware support where a counter is added to each cache line to track n and a small table is added to keep track of $N(cseq_X(d_X, n_X))$. We found that each counter only needs to be 7 bits because there are very few n_X values that are larger than 128.

Step 2 and 3: Computing $T(cseq_X(d_X, \overline{n_X}))$ and $E(n_Y)$

To compute the expected time interval duration for a circular sequence, we simply divide it by the access frequency per set of thread X (Af_X):

$$T(cseq_X(d_X, \overline{n_X})) = \frac{\overline{n_X}}{Af_X} \tag{2.5}$$

To estimate how many accesses by Y are expected to happen during the time interval $T(cseq_X(d_X, \overline{n_X}))$, we simply multiply it with the access frequency per set of thread

Y:

$$E(n_Y) = Af_Y \times T(cseq_X(d_X, \overline{n_X})) \tag{2.6}$$

Step 4: Computing $P(seq_Y(d_Y, E(n_Y)))$

The problem can be stated as finding the probability that given $E(n_Y)$ accesses from thread Y, there are d_Y distinct addresses, where d_Y is a random variable. For simplicity of Step 4's discussion, we will just write $P(seq(d,n))$ to represent $P(seq_Y(d_Y, E(n_Y)))$. The following theorem uses inductive probability function to compute $P(seq(d,n))$. The theorem assumes that each access is independent and identically distributed according to the overall stack distance profile.

Theorem 1 *For a sequence of n accesses from a given thread, the probability of the sequence to have d distinct addresses can be computed with a recursive relation, i.e.* $P(seq(d,n)) =$

$$\begin{cases} 1 & \textbf{if } n = d = 1 \\ P((d-1)^+) \times P(seq(d-1, d-1)) & \textbf{if } n = d > 1 \\ P(1^-) \times P(seq(1, n-1)) & \textbf{if } n > d = 1 \\ P(d^-) \times P(seq(d, n-1)) + & \\ \quad P((d-1)^+) \times P(seq(d-1, n-1) & \textbf{if } n > d > 1 \end{cases}$$

where $P(d^-) = \sum_{i=1}^{d} P(cseq(i,*))$ *and* $P(d^+) = 1 - P(d^-)$.

Proof: The proof will start from the more complex term to the least complex term.

Case 1 ($n > d > 1$): Let the sequence $seq(d, n)$ represent an access sequence $Y_1, Y_2, \ldots, Y_{n-1}, Y_n$. The sequence just *prior* to this one is $Y_1, Y_2, \ldots, Y_{n-1}$. There are two possible subcases. The first subcase is when the address accessed by Y_n also appears in the prior sequence, i.e. $addr(Y_n) \in \{addr(Y_1), addr(Y_2), \ldots, addr(Y_{n-1})\}$, hence the prior sequence is $seq(d, n-1)$. Furthermore, adding Y_n to the prior sequence creates a new circular sequence $cseq(i, *)$ with i ranging from 1 to d, with a probability of $1 - \sum_{i=d+1}^{\infty} P(cseq(i, *))$, which is equal to $\sum_{i=1}^{d} P(cseq(i, *))$, denoted as $P(d^-)$. The second subcase is when the address accessed by Y_n has not appeared in the prior sequence, i.e. $addr(Y_n) \notin \{addr(Y_1), addr(Y_2), \ldots, addr(Y_{n-1})\}$, hence the prior sequence is $seq(d-1, n-1)$. Furthermore, adding Y_n to the prior sequence does not create a new circular sequence at all (i.e. $cseq(\infty, *)$), or creates a circular sequence that is not within the sequence (i.e. $cseq(i, *)$ where $i > d - 1$). Therefore, the probability of the second subcase is $\sum_{i=d}^{\infty} P(cseq(i, *)) = 1 - \sum_{i=1}^{d-1} P(cseq(i, *))$, denoted as $P((d-1)^+)$ [3]. Therefore, $P(seq(d, n)) = P(d^-) \times P(seq(d, n-1)) + P((d-1)^+) \times P(seq(d-1, n-1))$.

Case 2 ($n > d = 1$): since $seq(1-1, n-1)$ is impossible, $P(seq(d-1, n-1)) = 0$. Therefore, $P(seq(1, n)) = (P(1^-) \times P(seq(1, n-1))$ follows from Case 1.

Case 3 ($n = d > 1$): since $seq(d, n-1)$ is impossible (there are more distinct addresses than accesses), $P(seq(d, n-1)) = 0$. Therefore, $P(seq(d, n)) = P((d-1)^+) \times P(seq(d-1, d-1))$ follows from Case 1.

[3]Computation-wise, $\sum_{i=d}^{\infty} P(cseq(i, *)) = \frac{C_{>A} + \sum_{i=d}^{i=A} C_i}{totAccess}$.

Case 4 ($n = d = 1$): $P(seq(1,1)) = 1$ is true because the first address is always considered distinct.

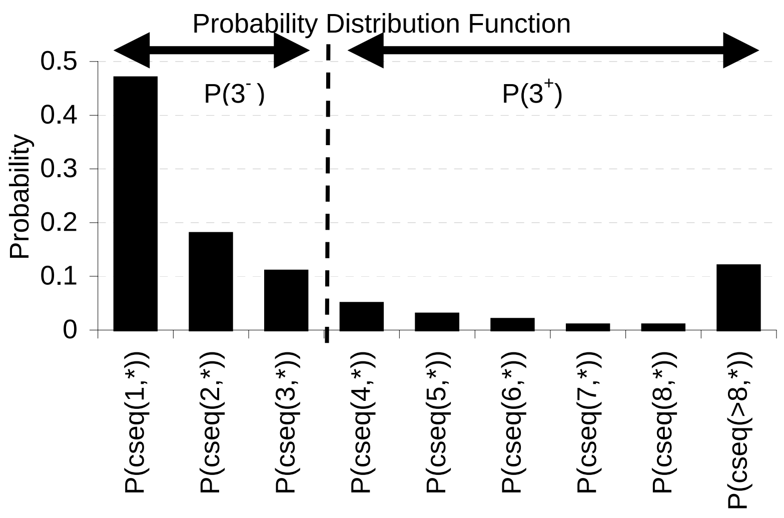


Figure 2.7: Example probability distribution function that shows $P(3^-)$ and $P(3^+)$. The function is computed by using the formula in Corollary 2.

Corollary 2 and Figure 2.7 illustrates how $P(d^-)$ and $P(d^+)$ can be computed from the stack distance profile. The figure shows that we already have three distinct addresses in a sequence. The probability that the next address will be one already seen is $P(3^-)$, otherwise it is $P(3^+)$.

Step 5: Computing the Number of Extra Misses Under Sharing

Step 4 has computed $P(seq_Y(d_Y, E(n_Y)))$ for all possible values of d_Y. We can then compute $P_{miss}(cseq_X(d_X, \overline{n_X}))$ using Equation 2.3. To find the total number of misses for thread X due to cache contention with thread Y, we need to multiply the probability of a cache miss from a particular circular sequence with the number of occurrences of such a circular sequence, then sum them over all possible values of d_X, and add the result to the original number of cache misses ($C_{>A}$):

$$miss_X = C_{>A} + \sum_{d_X=1}^{A} P_{miss}(cseq_X(d_X, \overline{n_X})) \times C_{d_X} \tag{2.7}$$

2.2 Circular Sequence Profiling

The inductive probability model (*Prob*) needs applications' circular sequence profiles as its input in Step 1 of the prediction model. A circular sequence profile essentially captures the temporal reuse behavior of an application. The information collected through circular sequence profiling is the superset of that collected through stack distance profiling. While stack distance profiling only collects the distribution of the number of distinct addresses that occur between two consecutive accesses to the same line, circular sequence profiling collects the distribution of the number of distinct and non-distinct addresses that occur between two consecutive accesses to the same line.

Although off-line profiling through simulation can be used to collect this information, this section shows that with a simple hardware support, we can also collect the information online with low overheads. While runtime circular sequence profiling is not strictly needed for constructing a performance model or for understanding the performance impact of cache sharing, there are situations in which it may be preferable compared to off-line profiling using a simulator. One such situation is when dynamic changes of an application's circular sequence profile needs to be captured for dynamic adaptation of the cache partitioning policy. Another situation is when simulation is expensive or impractical, such as for applications with large inputs or long-running computation.

Figure 2.8 shows the hardware support for circular sequence profiling. To collect

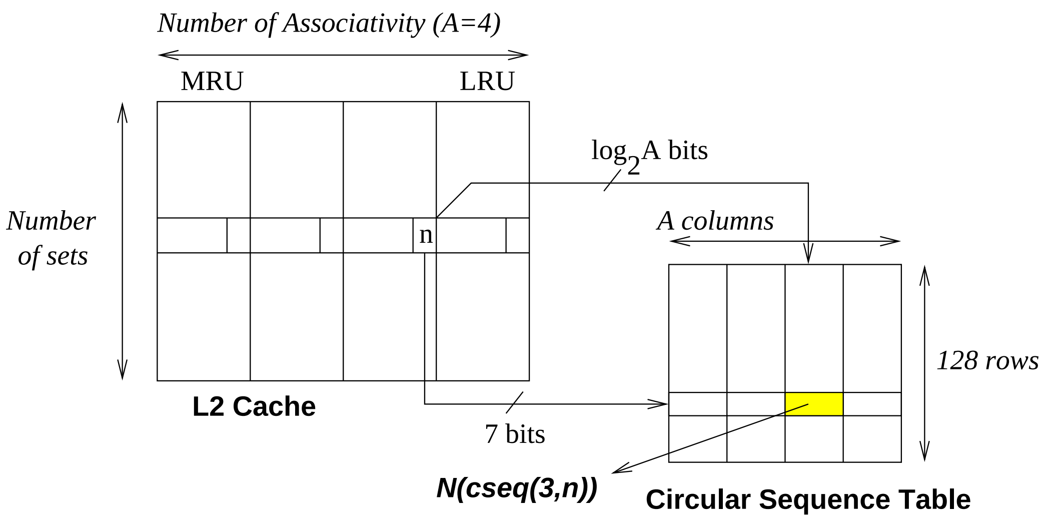


Figure 2.8: Hardware support for circular sequence profiling, illustrating a 4-way associative cache.

$N(cseq(d,n))$, we need to track both d and n for each cache access. For an LRU cache, the LRU stack position of a line when it is accessed provides d. To track n, each cache line can be augmented with a counter that is incremented on each access to any line in the same set. In practice, we find very few $n > 128$. Therefore a 7-bit counter is sufficient (if it is not sufficient, it can be truncated without much loss of information). Assuming the cache has 64-byte lines, this is equivalent to a storage overhead of 1.4% of the L2 cache. In addition, a *Circular Sequence Table* is added. The table has $2^7 = 128$ rows and A columns, where A is the cache associativity. Each entry holds a 32-bit counter, which is able to record a value up to 2^{32}, or more than 4 billion samples, allowing for a very long continuous profiling period. The table size is reasonably small, e.g., for a 4-way associative cache, the size is 2 Kbytes.

When a cache line is re-accessed, a circular sequence is detected. The position in the LRU stack (d) is used to index the column of the circular sequence table, the cache line's n-field is used to index the row of the table, and the corresponding entry in the table is incremented to reflect the current number of the circular sequence $cseq(d, n)$, i.e. $N(cseq(d, n))$. Then, the n-field is reset to 0, and the LRU stack is updated by the LRU replacement algorithm.

2.3 Validation Methodology

Simulation Environment. The evaluation is performed using a detailed CMP architecture simulator based on SESC, a cycle-accurate execution-driven simulator developed at the University of Illinois at Urbana-Champaign [23]. The CMP cores are out-of-order superscalar processors with private L1 instruction and data caches, and shared L2 cache and all lower level memory hierarchy components. Table 2.2 shows the parameters used for each component of the architecture. The L2 cache uses prime modulo indexing to ensure that the cache sets' utilization is uniform [24]. Unless noted otherwise, the L2 cache only stores data and does not store instructions.

Table 2.2: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip from the processor.

CMP
2 cores, each 4-issue dynamic. 3.2 GHz. Int, fp, ld/st FUs: 3, 2, 2
Branch penalty: 13 cycles. Re-order buffer size: 152
MEMORY
L1 Inst, Data (private): each WB, 32 KB, 4 way, 64-B line, RT: 2 cycles, LRU replacement
L2 data (shared): WB, 512 KB, 8 way, 64-B line, RT: 12 cycles, LRU replacement, prime modulo indexed, inclusive.
RT memory latency: 362 cycles
Memory bus: split-transaction, 8 B, 800 MHz, 6.4 GB/sec peak

Applications. To evaluate the benefit of the cache partitioning schemes, we choose a set of mostly memory-intensive benchmarks: *apsi*, *art*, *applu*, *equake*, *gzip*, *mcf*, *perlbmk* and *swim* from the SPEC2K benchmark suite [44]; and *mst* from the Olden benchmark suite. Table 2.3 lists the benchmarks, their input sets, and their L2 cache miss rates over the benchmarks' entire execution time. The miss rates may differ from when they are co-scheduled, because the duration of co-scheduling may be shorter than the entire execution of the benchmarks. These benchmarks are paired and co-scheduled. Fourteen benchmark pairs that exhibit a wide spectrum of stack distance profile mixes are evaluated.

Table 2.3: The applications used in our evaluation.

Benchmark	Input Set	L2 Miss Rate (whole execution)
art	test	99%
applu	test	68%
apsi	test	5%
equake	test	91%
mst	1024 nodes	63%
gzip	test	3%
mcf	test	9%
perlbmk	reduced ref	59%
swim	test	75%

Co-scheduling. Benchmark pairs run in a co-schedule until a thread that is shorter completes. At that point, the simulation is stopped to make sure that the statistics collected are only due to sharing the L2 cache. To obtain accurate stack distance or circular sequence profiles, for the shorter thread, the profile is collected for its entire execution without cache sharing, but for the longer thread, the profile is collected for the same number of instructions as that in the co-schedule.

2.4 Validation and Evaluation

This section will discuss four sets of results: the impact of cache sharing on IPC (Section 2.4.1), validation of the prediction models (Section 2.4.2), sensitivity study (Section 2.4.4) and a case study on the relationship between temporal reuse behavior and the impact of cache sharing (Section 2.4.5).

2.4.1 Impact of Cache Sharing

Figure 2.9 shows the impact of cache sharing on IPC of each benchmark in a co-schedule. Each group of two bars represents a co-schedule consisting of two threads from sequential benchmarks that run on different CMP cores. The full height of each bar (black + white sections) represents the IPC of the benchmark when it runs alone in the CMP. The black section represents the IPC of the benchmark when it is co-scheduled with another benchmark. Therefore, the white section represents the reduction in IPC of the benchmark due to L2 cache sharing.

There are several interesting observations that can be made from the figure. First, the figure confirms that the impact of cache sharing is neither uniform nor consistent across benchmarks. For most co-schedules, the IPC reduction of the benchmarks is highly non-uniform. For example, in *applu+art*, while *applu* suffers from 42% IPC reduction, *art* only suffers 14% IPC reduction. Similar observation can be made for *applu+equake* (6% vs. 19%), *art+equake* (10% vs. 41%), *gzip+applu* (25% vs. 4%), *gzip+apsi* (20% vs. 0%), *mcf+art* (65% vs. 10%), and many others. In addition, for

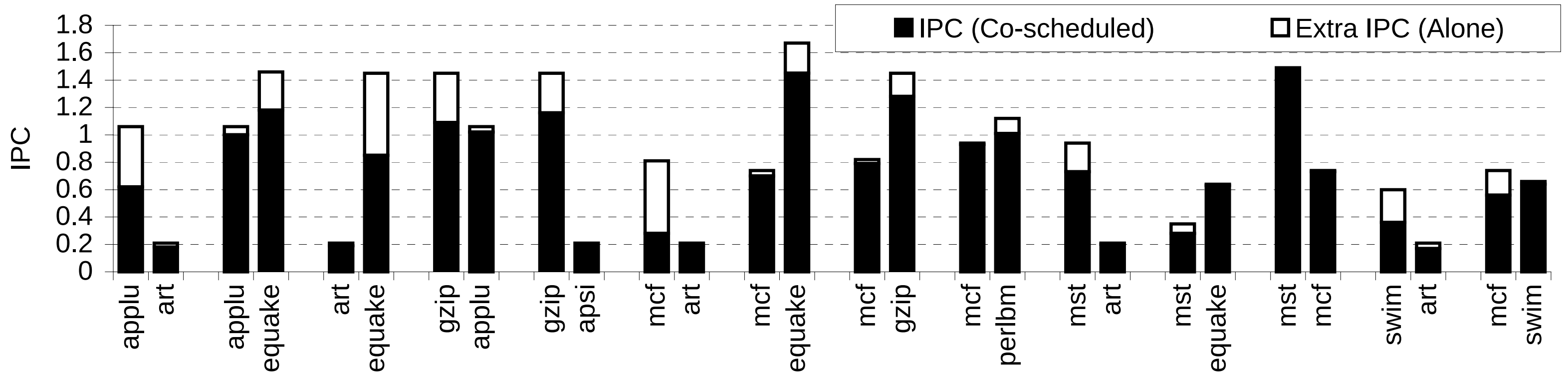


Figure 2.9: The impact of cache sharing on the IPCs of co-scheduled threads.

almost all benchmarks, the IPC reduction is not consistent for the same benchmark across different co-schedules. For example, the IPC reduction for *equake* is 19% in *applu+equake*, 41% in *art+equake*, 13% in *mcf+equake*, and 2% in *mst+equake.* The same observation can be made for *applu*, *mcf*, *gzip*, and *swim.* Another observation is that a few benchmarks, such as *apsi* and *art*, do not exhibit much slowdown due to cache sharing. They are applications with very low IPC values because they suffer many L2 cache misses, where most of them are capacity misses. Therefore, even when the effective cache space decreases, the number of cache misses cannot increase much.

2.4.2 Model Validation

Table 2.4 shows the validation results for the fourteen co-schedules that we evaluate. The first numeric column shows the number of instructions that each thread executes in a co-schedule. The second column denotes the extra L2 cache misses under cache sharing, divided by the L2 cache misses when each benchmark runs alone (e.g. 100% means that the number of cache misses under cache sharing is two times compared to when the benchmark runs alone). The cache misses are collected using simulation.

The next three columns present the prediction errors of the FOA, SDC, and *Prob* models. The last two columns (*Prob+NI*) and (*Prob+Unif*) will be discussed in Section 2.4.3. The errors are computed as the difference in the L2 cache misses predicted by the model and collected by the simulator under cache sharing, divided by the

Table 2.4: Validation Results. In each co-schedule, the benchmark that finishes to completion is indicated with an asterisk.

Co-schedule		Number of Instructions	Extra L2 Cache Misses	L2 Cache Miss Prediction Error (E_j)				
				FOA	SDC	*Prob*	*Prob+ NI*	*Prob+ Unif*
applu	applu	424M	29%	8%	-22%	2%	2%	7%
+art	art*	121M	0%	0%	0%	0%	0%	0%
applu	applu*	447M	10%	0%	1%	1%	0%	0%
+equake	equake	529M	19%	6%	1%	5%	7%	1%
art	art*	121M	0%	0%	0%	0%	0%	0%
+equake	equake	546M	43%	-6%	-30%	5%	8%	4%
gzip	gzip*	287M	243%	-60%	-58%	-25%	-26%	-35%
+applu	applu	269M	11%	6%	4%	2%	2%	5%
gzip	gzip*	287M	180%	-62%	-64%	-9%	-9%	-11%
+apsi	apsi	52M	0%	0%	0%	0%	0%	0%
mcf	mcf	177M	296%	-4%	-74%	7%	12%	4%
+art	art*	121M	0%	0%	0%	0%	0%	0%
mcf	mcf*	187M	11%	-9%	-3%	-3%	-3%	-6%
+equake	equake	388M	6%	22%	7%	5%	6%	4%
mcf	mcf	176M	18%	-5%	1%	7%	7%	6%
+gzip	gzip*	287M	102%	264%	25%	22%	22%	21%
mcf	mcf	159M	8%	-5%	-7%	-3%	-3%	-11%
+perlbmk	perlbmk*	174M	28%	30%	31%	2%	-3%	4%
mst	mst	450M	10%	0%	-5%	0%	0%	-1%
+art	art*	121M	0%	0%	0%	0%	0%	0%
mst	mst	530M	25%	4%	4%	3%	3%	2%
+equake	equake*	1185M	3%	1%	-2%	0%	0%	0%
mst	mst	382M	0%	0%	0%	0%	0%	-1%
+mcf	mcf*	187M	2%	0%	-2%	0%	0%	0%
swim	swim	261M	0%	0%	0%	0%	0%	0%
+art	art*	121M	0%	0%	0%	0%	0%	0%
mcf	mcf*	187M	59%	-31%	-32%	-7%	-7%	-8%
+swim	swim	213M	0%	0%	0%	0%	0%	0%
Minimum Absolute Error ($min(\|E_j\|)$)				0%	0%	0%	0%	0%
Maximum Absolute Error ($max(\|E_j\|)$)				264%	74%	25%	26%	35%
Arithmetic Mean of Absolute Error				**18.6%**	**13.2%**	**3.9%**	4.3%	4.7%
Geometric Mean of Absolute Error				**13.3%**	**11.6%**	**3.7%**	4.1%	4.4%

number of L2 cache misses collected by the simulator under cache sharing. Therefore, a positive number means that the model predicts too many cache misses, while a negative number means that the model predicts too few cache misses. The last four rows in the table summarize the errors. They present the minimum, maximum, arithmetic mean, and geometric mean of the errors, after each error value is converted to its absolute (positive) value.

Consistent with the observation of IPC values in Section 2.4.1, the benchmarks that show large IPC reduction also suffer from many extra L2 cache misses, with one exception. Specifically, the IPC reduction for *swim* in *swim+art* is not caused by an increase in cache misses. Rather, it is caused by memory bandwidth contention that results in higher cache miss penalties. In five co-schedules, one of the benchmarks suffers from 59% or more extra cache misses: *gzip+applu* (243% extra misses in *gzip*), *gzip+apsi* (180% extra misses in *gzip*), *mcf+art* (296% extra misses in *mcf*), *mcf+gzip* (102% extra misses in *gzip*), and *mcf+swim* (59% extra misses in *mcf*).

Let us compare the average absolute prediction error of each model. *Prob* achieves the highest accuracy, followed by SDC and FOA (average error of 3.9% vs. 13.2% vs. 18.6%, respectively). The same observation can be made when comparing the maximum absolute errors: 25% for *Prob*, 74% for SDC, and 264% for FOA. Therefore, *Prob* achieves a substantially higher accuracy compared to both SDC and FOA.

Analyzing the errors for different co-schedules, *Prob*'s prediction errors are larger than 10% only in two cases where a benchmark suffers a very large increase in cache misses, such as *gzip* in *gzip+applu* (-25% error, 243% extra cache misses), and *gzip* in *mcf+gzip* (22% error, 102% extra cache misses). Since the model still correctly identifies a large increase in the number of cache misses, it is less critical to predict such cases very accurately. Elsewhere, *Prob* is able to achieve a very high accuracy, even in cases where there is a large number of extra cache misses. For example, in *mcf+art*, *mcf* has 296% extra cache misses, yet the error is only 7%. In *mcf+swim*, *mcf* has 59% extra cache misses, yet the error is only -7%. Finally, in *gzip+apsi*, *gzip* has 180% error, yet the error is only -9%.

In general, both FOA and SDC are not as accurate as *Prob*, although SDC performs better than FOA, with an average absolute error of 13.2% (vs. 18.6% for FOA), and maximum absolute error of 74% (vs. 264% for FOA). Unfortunately, the large error not only happens in cases where the extra number of cache misses is large, but also in cases where the extra number of cache misses is small. For example, in *mcf+perlbmk*, *perlbmk* has 28% extra cache misses, and the prediction error is 30% for FOA and 31% for SDC.

2.4.3 Remaining Inaccuracy

To further validate the *Prob* model, we relax two assumptions that we have made in Section 2.1.1, namely the multi-level cache inclusion, and the unified L2 cache. The last two columns in Table 2.4 shows the prediction error of *Prob* when the L2 cache does not maintain inclusion with the L1 data cache (*Prob+NI*), and when the

L2 cache stores both instructions and data (*Prob+Unif*). In both cases, we rerun the simulation and the profiling, and generate new predictions.

For an inclusive L2 cache, the effect of inclusion is ignored by our models. When an inclusive L2 cache replaces a cache line, the corresponding line in the L1 cache is invalidated. This may cause extra L1 cache misses that perturb the L2 accesses and miss rates, which the models assume to be unchanged. In *Prob+NI*, we simulate a non-inclusive L2 cache, thereby removing one source of possible inaccuracy. The result in *Prob+NI* shows that the impact of cache inclusion property is insignificant. The average error increases by only 0.4% to 4.3%.

Using an L2 cache that stores both data and instructions, the prediction error in *Prob+Unif* increases by 0.8% to 4.7%. In some co-schedules, the errors increase slightly, and in others, the errors decrease slightly. We conclude that the increase in prediction errors only matter for a small subset of co-schedules, because in most benchmarks, the instruction footprint is a lot smaller than the data footprint.

Prob's remaining inaccuracy may be due to two assumptions. We assume that the number of accesses in a circular sequence of a thread X can be represented accurately by its expected value ($\overline{n_X}$ in Equation 2.4). We also assumed that the number of accesses from an interfering thread Y can be represented accurately by its expected value ($E(n_Y)$ in Equation 2.6). In addition, the model rounds down $E(n_Y)$ to the nearest integer. Relaxing these assumptions requires treating n_X and n_Y as random variables, which significantly complicates the *Prob* model.

2.4.4 Sensitivity Study

To observe the impact of L2 cache parameters to the prediction accuracy of the *Prob* model, we perform two studies. In the first study, we vary the L2 cache size from 256KB to 1024KB, while keeping the associativity at 8-way. In the second study, we vary the L2 cache associativity from 4 to 16, while keeping the cache size constant at 512KB. The results are shown in Table 2.5 and Table 2.6, respectively.

Table 2.5: Prediction accuracy for different cache sizes.

Cache Size, Assoc		Extra L2 Cache misses	L2 Miss Prediction Absolute Error (E_j)		
			FOA	SDC	*Prob*
256KB, 8	*min*	0%	0%	0%	0%
	max	4332%	54%	153%	31%
	avg	283%	9.6 %	27.3%	4.2%
512KB, 8	*min*	0%	0%	0%	0%
	max	296%	264%	74%	25%
	avg	39%	18.6 %	13.2%	3.9%
1024KB, 8	*min*	0%	0%	0%	0%
	max	363 %	74%	78%	21%
	avg	45%	17.7 %	21.1%	5.4%

An interesting observation is that for a 256KB L2 cache, the average and the maximum increase in L2 cache misses is now 283% and 4332%. Therefore, the impact of cache sharing for a small cache is very significant. In terms of the average error of *Prob* for different L2 cache sizes, Table 2.5 shows little variation (4.2% for 256KB, 3.9% for 512KB, and 5.4% for 1MB). However, the error tends to be correlated with the increase in the number of L2 cache misses. As discussed earlier, *Prob*'s large prediction errors only occur when there are large increases in L2 cache misses. Since, in both 256KB and 1MB cache sizes, the average increase in L2 cache misses is larger than that in the 512KB cache, the prediction errors are also larger. However, the maximum absolute error reveals a different trend. The maximum error decreases with a larger L2 cache size (31% for 256KB, 25% for 512KB, and 21% for 1MB), indicating that *Prob* is more accurate in the worst case for larger caches. In any case, *Prob* achieves very good accuracy.

Both the FOA and SDC models have large errors. For 256KB cache, SDC performs very poorly, with an average error of 27.3% and maximum absolute error of 153%. This is expected as SDC does not take into account the miss frequency of the benchmarks, which tends to increase with smaller caches.

Table 2.6: Prediction accuracy for different cache associativities.

Cache Size, Assoc		Extra L2 Cache misses	L2 Miss Prediction Absolute Error (E_j)		
			FOA	SDC	*Prob*
512KB, 4	*min*	0%	0%	0%	0%
	max	361%	80%	78%	45%
	avg	47%	12.2 %	13.4%	7.2%
512KB, 8	*min*	0%	0%	0%	0%
	max	296%	264%	74%	25%
	avg	39%	18.6 %	13.2%	3.9%
512KB, 16	*min*	0%	0%	0%	0%
	max	275%	69%	73%	36%
	avg	38.1%	11.8 %	12.5%	5.1%

Table 2.6 shows a decreasing average and maximum L2 cache miss increase as the

associativity increases, indicating that higher associativity can tolerate cache sharing impact better. In terms of the average error of *Prob*, there is little variation (7.2% for 4-way, 3.9% for 8-way, and 5.1% for 16-way associativity). For a small associativity (4-way), the larger error is expected because we use $E(n_Y)$ instead of treating n_Y as a random variable. Unfortunately, for smaller associativity, the value of $E(n_Y)$ tends to be smaller too. Since we round it down to the nearest integer value, this rounding error starts to introduce extra inaccuracy. For 16-way associativity, the prediction error is slightly higher than in the 8-way associativity mostly due to a larger error in one co-schedule. To summarize, for sufficiently high associativity, *Prob* remains very accurate.

2.4.5 Case Study: Impact of Temporal Reuse Behavior on Cache Sharing Performance

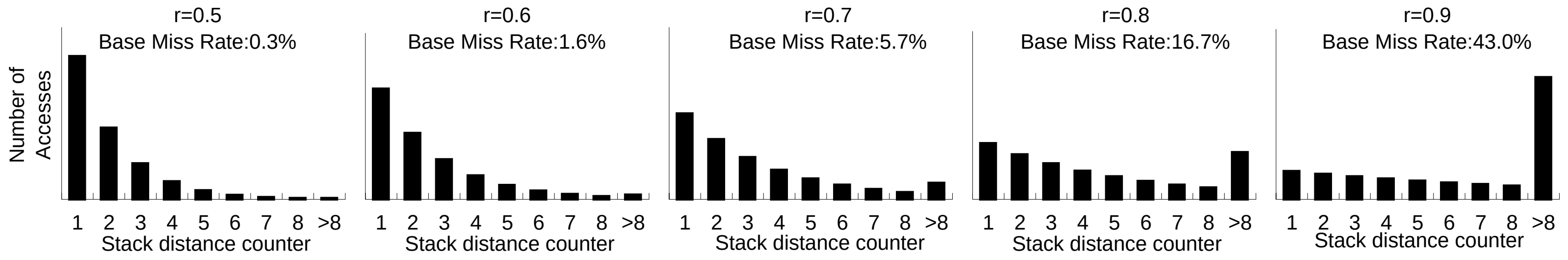
In this case study, we investigate how temporal reuse behavior affects the cache sharing performance of co-scheduled applications. To achieve that, we construct synthetic stack distance profiles that cover a wide range of temporal reuse behavior. The synthetic stack distance profiles are constructed using geometric progressions, where $C_1 = Z, C_2 = Zr, C_3 = Zr^2, \ldots, C_A = Zr^{A-1}$, where Z denotes the *amplitude*, and $r = 0.5, 0.6, \ldots, 0.9$ denotes the *common ratio* of the progression. In order to assign a value to the miss counter ($C_{>A}$), we use two models. The first (*full geometric*) model assumes that the geometric progression continues infinitely, hence

$C_{>A} = \sum_{i=A+1}^{\infty} Zr^i = \frac{Zr^A}{1-r}$. In practical terms, this model assumes that any line that has been accessed has a probability of being reused, and its probability decreases according to its stack position infinitely. Figure 2.10(a) shows the full-geometric synthetic stack distance profiles as the common ratio is varied from 0.5 to 0.9. The figure shows that larger common ratio indicates a flatter profile, while a smaller ratio indicates a more concentrated profile.

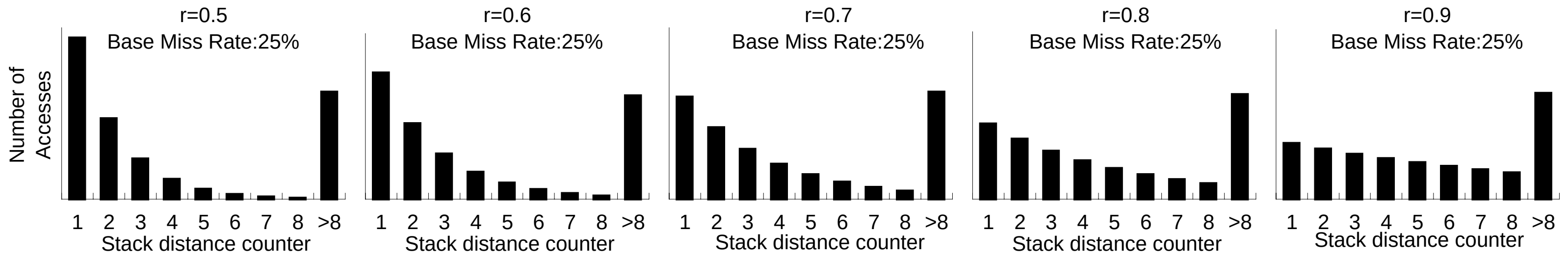
The second (*partial-geometric*) model assumes that temporal reuse does not continue infinitely and the miss rate is chosen as a constant fraction of the total accesses. For example, Figure 2.10(b) shows the partial-geometric synthetic stack distance profiles with a fact that $C_{>A}$ equals to 25% of the total accesses.

Modeling a stack distance profile as a geometric progression is in general reasonable because recently used lines are more likely to be reused than less recently used lines, a well-known behavior of applications exploited by the LRU replacement policy. However, how much temporal reuse an application has may determine whether its stack distance profile will be closer to the full- or partial-geometric model.

To analyze the cache sharing impact for various profiles, we start with the same profile for both a *base thread* and an *interfering thread*. Then, we modify the interfering thread's profile by four different ways and observe how the number of misses of the base thread is impacted. Let C_{total} denote the total number of accesses of the base thread, i.e. $C_{total} = C_1 + C_2 + \ldots + C_A + C_{>A}$. We define a multiplying factor $k = 0.1, 0.2, \ldots, 0.5$ and multiply it with C_{total} to obtain the extra accesses that we



(a)



(b)

Figure 2.10: The full-geometric stack distance profiles with different common ratios (a), and the partial-geometric stack distance profiles with different common ratios, with a miss rate fixed at 25%(b).

will add to the interfering thread's profile as shown in Figure 2.11. We can add the extra accesses solely to the miss counter (Excess$C_{>8}$), the MRU counter (ExcessC_1), or across all counters while maintaining its common ratio constant (ExcessC_*). The final modification we make to the interfering thread is performed without increasing its number of accesses, but changing its common ratio to $r' = 0.5, 0.6, \ldots, 0.9$. These four modifications to the interfering thread's profile are applied in the same way to both the full-geometric and partial-geometric models. For all modifications, we assume that the execution time of the interfering thread is unchanged, i.e. the change in the number of access proportionally changes the access frequency.

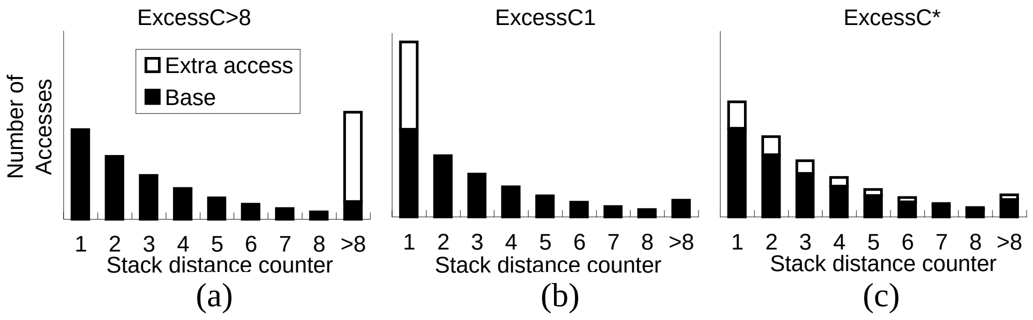


Figure 2.11: Example modifications to the interfering thread's stack distance profile over a full-geometric base stack distance profile with a common ratio of 0.7.

Note that the *prob* model expects the circular sequence profiles as its input rather than stack distance profiles, requiring the value of $\overline{n}$ for each d. For simplicity, we set $\overline{n} = d + 1$, which, in our experience, is in line with many (but not all) applications.

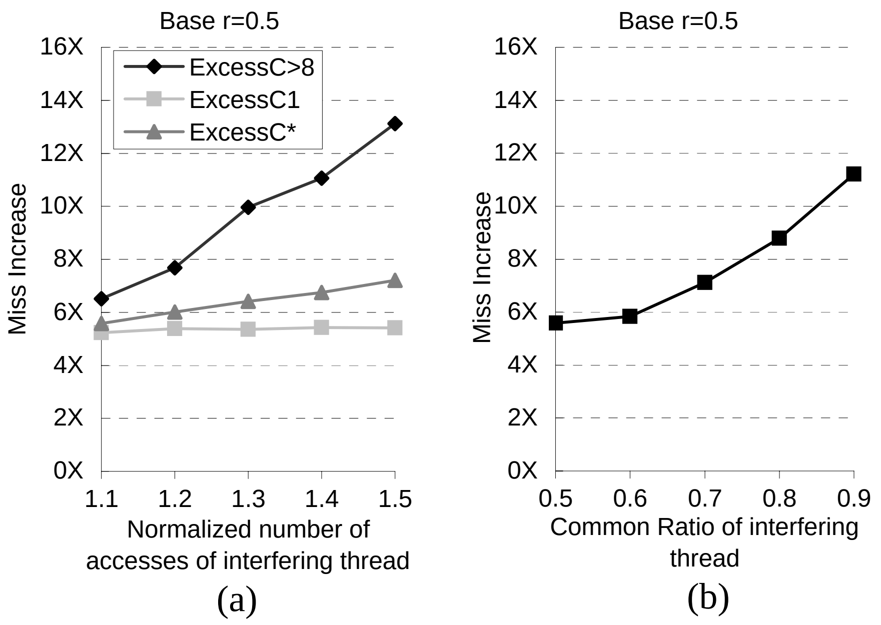


Figure 2.12: The impact of cache sharing to a full-geometric base thread with a common ratio of $r = 0.5$.

Results for Full-Geometric Model

Figure 2.12 shows the result of the four experiments when the base thread's common ratio is 0.5. The x-axes of Figure 2.12(a) shows the number of accesses of the interfering thread relative to the base thread, while the y-axes shows the miss increase. Figure 2.12(b) shows the miss increase of the base thread when the interfering thread has different common ratios but without extra accesses.

We can observe in Figure 2.12(a) that Excess$C_{>8}$ incurs the highest miss increase on the base thread, followed by ExcessC_* and ExcessC_1. This is because Excess$C_{>8}$ greatly expands the working set size of the interfering thread, reducing the effective cache space of the base thread. On the other hand, ExcessC_1 only increases the reuse of the interfering thread's MRU lines, which are likely to already be a part of the interfering thread's effective cache space. As a result, the base thread does not suffer from much miss increase. Thus, we can expect that extra reuses to a larger stack position in the interfering thread are more harmful to the base thread. In addition, we can also observe that a base thread with a small common ratio, indicating a concentrated stack distance profile, is very vulnerable to miss increase due to cache sharing, ranging from 5-13 times the base misses.

From Figure 2.12(b), we can make an observation that the larger common ratio of the interfering thread (i.e. the flatter the stack distance profile), the higher the miss increase of the base thread. This is very significant considering that the access numbers of both threads are identical. The reason for this is that with a flatter stack distance profile, the interfering thread reuses and retains the cache lines it has brought into the cache, effectively reducing the space available to the base thread.

Overall, we identify that the base misses and stack distance profile shape of the interfering thread play a significant role in affecting the base thread's miss increase.

Figure 2.13 and Figure 2.14 repeat the experiments in Figure 2.12 for the base thread's common ratios of 0.7 and 0.9, respectively. The figures reveal that the higher common ratio in the base thread, the smaller miss increase the base thread suffers. In fact, when the common ratio is 0.9, the base thread's miss increase never rise above 1X (or 100%). This is not unexpected since the base misses for a large common ratio are also large, placing a smaller upper bound on the miss increase.

Overall, Figure 2.12– 2.14 help explain why *gzip* and *mcf* are very vulnerable to

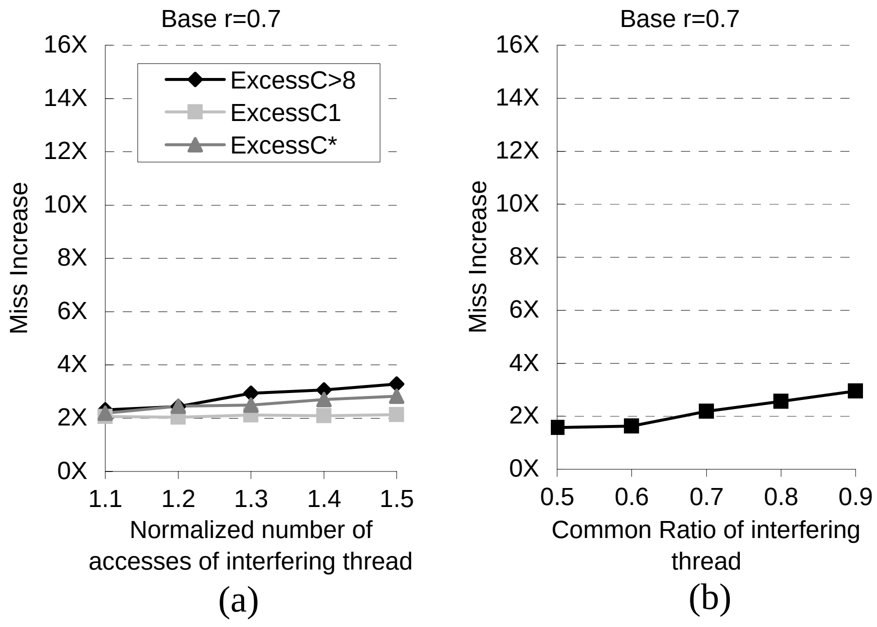


Figure 2.13: The impact of cache sharing to a full-geometric base thread with a common ratio of $r = 0.7$.

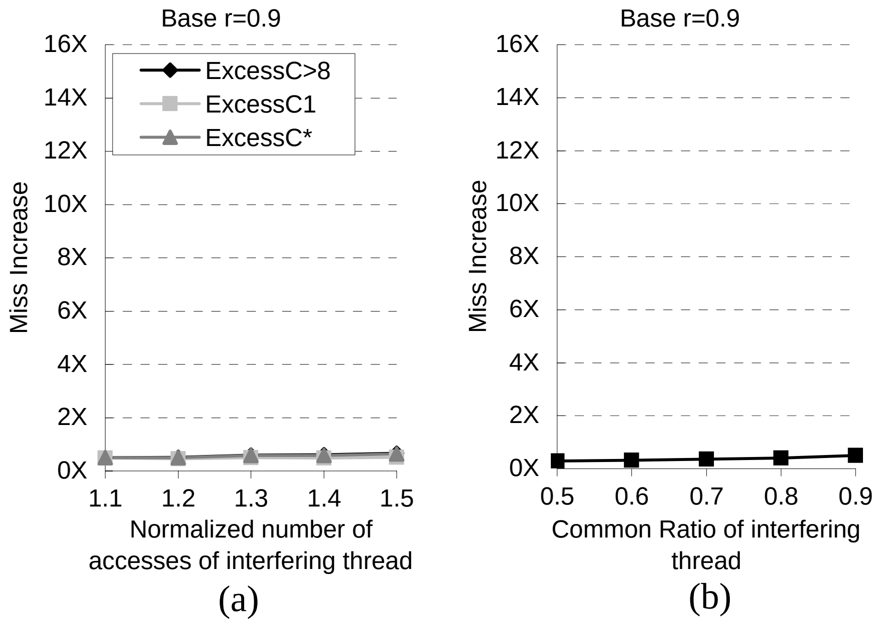


Figure 2.14: The impact of cache sharing to a full-geometric base thread with a common ratio $r = 0.9$.

a large miss increase due to cache sharing: they have low base misses, while almost all other applications that they are co-scheduled with have higher base misses and flatter stack distance profiles.

Results for Partial-Geometric Model

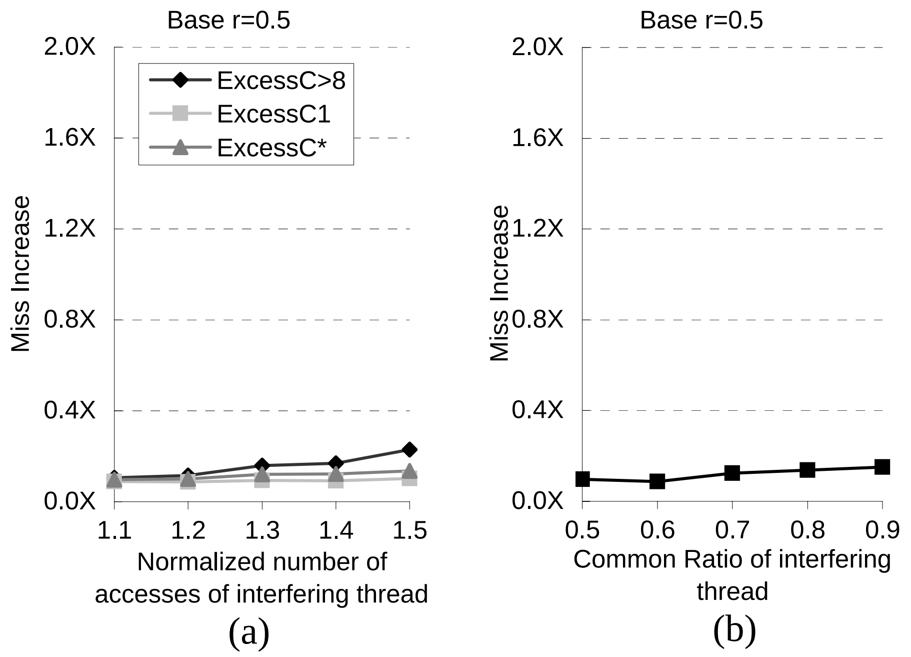


Figure 2.15: The impact of cache sharing to a partial-geometric base thread with a common ratio of $r = 0.5$.

Figure 2.15– 2.17 repeat the experiments in Figure 2.12– 2.14 for the partial-geometric model. A new striking observation is that whereas in the full-geometric model a base thread with a smaller common ratio is more vulnerable to cache miss increase, we observe the opposite in the partial-geometric model: the base thread's miss increase is up to 1.5X when its common ratio is 0.9 (Figure 2.17(a)), but only up to 0.3X when its common ratio is 0.5 (Figure 2.15(a)). The explanation for this

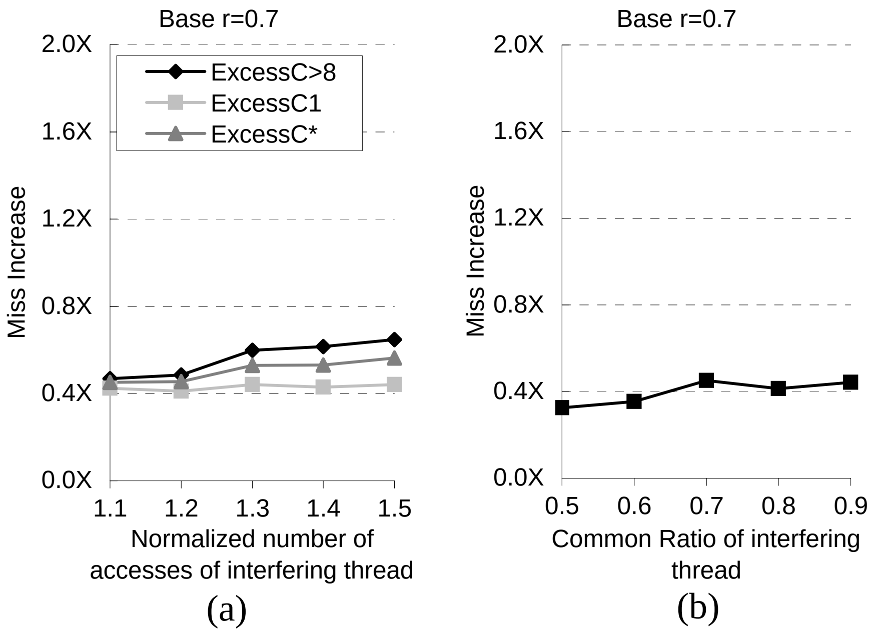


Figure 2.16: The impact of cache sharing to a partial-geometric base thread with a common ratio of $r = 0.7$.

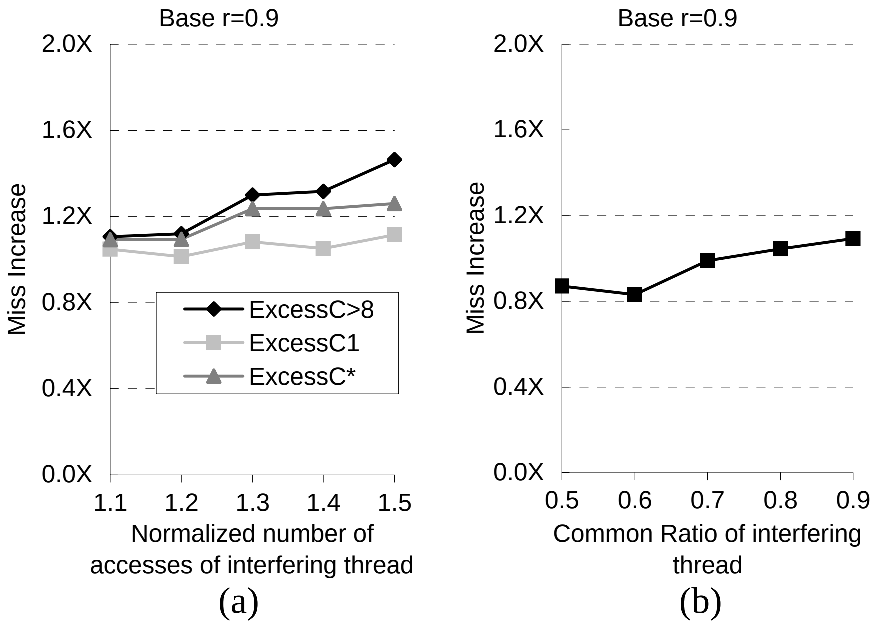


Figure 2.17: The impact of cache sharing to a partial-geometric base thread with a common ratio $r = 0.9$.

seemingly opposite behavior is that the partial-geometric model assumes that the base miss rates are constant across different common ratios and independent of the reuse probabilities at large stack distance positions. Given a constant base miss rate across different common ratios, we can conclude that a thread that has a flatter stack distance profile is more vulnerable to cache miss increase due to cache sharing. However, in the full-geometric model, such a flatter profile also has a high base miss rate, which more than offsets its vulnerability.

Finally, we note that because in the partial-geometric model the base miss rate is a constant percentage of the total accesses, the value of the constant greatly affects the miss increase. Hence, the magnitudes of miss increase for different constants can be very different than those in Figure 2.15– 2.17. For that reason, we do not compare the miss increase in the full- and partial-geometric models.

2.5 Related Work

Previous performance prediction models only predict the number of cache misses in a uniprocessor system [6, 9, 13, 14, 16, 26, 46, 52, 55], or predict cache contention on a single processor time-shared system [1, 49, 50]. In such a system, since only one thread runs at any given time, no interfering effects between threads in the cache is modeled. In contrast, this dissertation presents models that predict the impact of inter-thread cache sharing on each co-scheduled thread that shares the cache. As a result, the model can explain cache contention phenomena that have been observed in an SMT or CMP system in past studies [19, 24, 28, 48, 53], but have not been understood well.

The *Prob* model presented here may be applicable for improving OS thread scheduling decisions. For example, Snavely et al. rely on discovering the interaction (symbiosis) between threads in a SMT system by profiling all possible co-schedules [42]. Such profiling is unfeasible, or at least impractical, to implement on a real system due to the combinatoric explosion of the number of co-schedules that need to be profiled. If a symbiotic job scheduling is to be applied in a CMP system, *Prob* can avoid the need for such profiling by discovering cache symbiosis between co-scheduled threads without running the co-schedule, by directly using the model or through parameterization similar to the case study presented in Section 2.4.5.

Suh et al. [48] and Kim et. al. [25] have proposed partitioning the shared cache in a CMP system to minimize the number of cache misses or maximizing fairness. Both studies assume that a co-schedule is already determined by the OS, and the hardware's task is to optimize the performance for the given co-schedule. Unfortunately, some problems such as cache thrashing can only be avoided by the OS's judicious co-schedule selection. We view that our models can be used in a complementary way, where the models can be used to guide the OS scheduling algorithms, allowing them to optimize the performance of a selected co-schedule further.

In the context of paging behavior, Turner and Strecker showed a method to arrive at a closed-form solution for an equation similar to the one in Theorem 1 [51].

Finally, the model proposed by Wasserman et al. predicts the average cache miss penalty of a program on a superscalar uniprocessor [55], while that proposed by Solihin et al. predicts the miss penalty on a distributed shared memory multiprocessor system [43]. Integrating such models with *Prob* allow a full performance model that predicts the execution time rather than just miss rates.

2.6 Conclusions

This chapter has investigated the impact of inter-thread cache contention on a Chip Multi-Processor (CMP) architecture. Using a cycle-accurate simulation, we found that cache contention can significantly increase the number of cache misses of a thread in a co-schedule and showed that the degree of such contention is highly dependent on the thread mix in a co-schedule. We have proposed and evaluated two heuristics-based models and one analytical model that predict the impact of cache sharing on co-scheduled threads. The input to our models is the isolated L2 cache stack distance or circular sequence profiles of each thread, which can be easily obtained on-line or off-line. The output of the models is the extra number of L2 cache misses of each thread that shares the cache. We validated the models against a cycle-accurate simulation that implements a dual-core CMP architecture and found that the analytical Inductive Probability (*Prob*) model produces very accurate prediction regardless of the co-schedules and the cache parameters, with an average error of only 3.9% on a 512KB 8-way associative L2 cache. Finally, the *Prob* model provides a valuable and practical tool through which we can study the impact of cache sharing extensively. We have presented a case study to demonstrate how different temporal reuse behavior in applications influence the impact of cache sharing suffered by them. Through the case study, the *Prob* model reveals non-obvious interaction between the applications. We found that for a base thread, the base misses and stack distance profile shape of the interfering thread play a significant role in affecting the base thread's miss increase. We also found that a thread that has a flatter stack distance profile is more vulnerable to cache miss increase.

Chapter 3

Providing Performance Quality of Service in Chip Multi-Processors

This chapter is organized as follows. Section 3.1 presents QoS target specification and execution modes. Section 3.2 presents our microarchitecture technique for improving throughput. Section 3.3 presents improved resource stealing techniques for more effectively stealing resource. Section 3.4 discusses the implementation of an admission control policy. Section 3.5 presents the evaluation setup, while Section 3.6 presents and discusses the evaluation results. The related works are discussed in Section 3.7. Finally, Section 3.8 summarizes the findings.

3.1 QoS Target Specification and Execution Modes

This section presents the assumptions of our working environment (Section 3.1.1), QoS goals and metrics for specifying a QoS target (Section 3.1.2) and the proposed QoS execution modes (Section 3.1.3).

3.1.1 Definitions and Assumptions

We refer to a *job* as the unit of aperiodic computation task that has its own QoS target. A job may consist of a thread, an application, or a group of applications. In this dissertation, we limit our study by associating an instance of a single-threaded application as a job. We define *computation capacity* as the resources that can be used for providing performance. Basically, a job's QoS target is computation capacity *demand*, while available resources in the server are computation capacity *supply*.

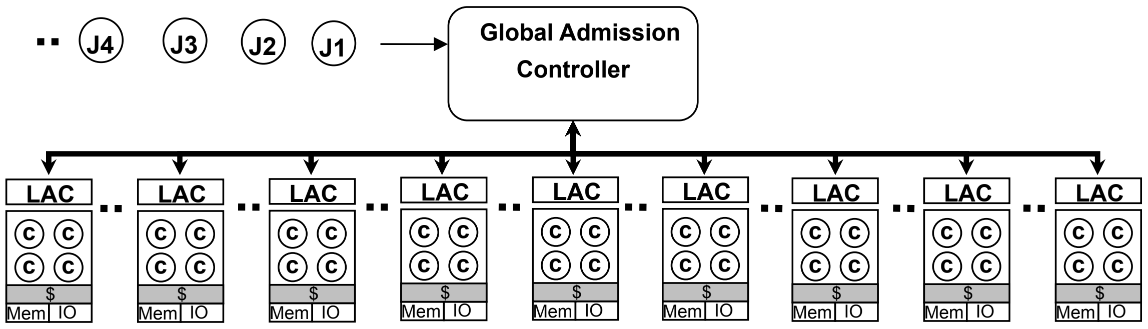


Figure 3.1: The assumed working environment.

We assume a server platform consisting of CMP nodes as shown in Figure 3.1. The server has a *global admission controller* (GAC) which decides whether to accept or reject a newly arriving job submitted by user. To achieve this, the global admission controller probes each CMP node's *local/per-CMP admission controller* (LAC) to find which CMP node can accept the job and satisfy its QoS target. When the GAC cannot find any CMP node that can accept the job, it rejects this job or negotiates with the user for another acceptable QoS target. A comprehensive discussion of the GAC is beyond the scope of this dissertation. In this dissertation, only the LAC is considered as a component of our QoS framework because it has a direct interaction with microarchitecture resources.

3.1.2 Specifying QoS Target

Earlier we have argued that for a CMP to fully provide QoS to an incoming job, *it must have available computation capacity in excess of the capacity demanded by the job.* We refer to the *units of target* as the units in which a QoS target is specified, and *units of capacity* as the units in which computation capacity in the CMP is expressed. Before continuing our discussion, it is helpful to define one term.

Definition 1 *A QoS target is* **convertible** *if its units of target can be converted into units of computation capacity.*

In order for a CMP to really provide QoS, two conditions must be met. The first condition is that a QoS target must be convertible, which allows the system to easily compare the available computation capacity with demanded capacity. The second condition is that a job should be accepted only if its QoS target can be satisfied. Convertibility and satisfiability checking are the basis for constructing an admission control policy to ensure that the QoS targets of *all* accepted jobs can be met.

One way to provide QoS in a CMP system is to model it after a traditional real-time system in which the QoS target of a job is specified by its *deadline*. In a real-time system, deadline convertibility can be achieved through *Worst-Case Execution Time* (WCET) analysis which determines the maximum execution time of a job by taking into account the maximum path length of the code and maximum latencies that can occur in the architecture. Unfortunately, in traditional real-time systems the operating system and processor architecture are often structured to suit the needs of real-time constraints, such as by restricting out-of-order execution, dynamic branch prediction, and a complex memory hierarchy. Our goal is to provide QoS in general purpose servers with a largely unmodified OS, processor architecture, and memory hierarchy. Furthermore, in a server environment, jobs often have unpredictable arrivals, dynamic and input-dependent behavior, and may not have meaningful deadlines. Hence, unlike in traditional real-time systems, we cannot use deadlines as the primary QoS target.

In [22], Iyer et al. proposed three types of QoS targets. The first is *Resource Usage Metrics (RUM)*, which specify the amount of resources needed by the application, such as the processor count, cache size, and bandwidth rate. The second is *Resource Performance Metrics (RPM)*, which specify the performance of specific resources being used, for example cache miss rates. The final is *Overall Performance Metrics (OPM)*, which specify the overall throughput of the program, expressed in IPC. Most prior studies in architecture support for QoS assume that the QoS target is expressed in IPC. However, we believe that IPC is not suitable to specify a QoS target because IPC is not easily convertible. A CMP system cannot easily determine how much IPC it can provide for a particular job (unless it uses an elaborate performance model). Furthermore, it also cannot easily determine the amount of platform resources that are needed to achieve a target IPC. Similarly, a CMP cannot easily determine what miss rate it can provide to a particular job or the amount of resources needed in order to provide a given miss rate. In fact, in addition to being non-convertible, it may be hard to check whether particular RPM and OPM values are *ill-defined*, i.e. they cannot be satisfied no matter how many resources are allocated. As a result, we believe that OPM and RPM are not suitable for a CMP system to fully provide QoS.

In contrast to RPM and OPM, RUM are easily convertible if a CMP is equipped with relatively simple hardware that tracks the current allocation of platform resources for different cores. For example, with RUM, an incoming job's request in terms of the amount of cache capacity it needs (demand) can be compared trivially against the amount of cache capacity that has not been allocated yet (supply). This leads to the ease of constructing an admission control policy. An additional benefit of using RUM is that such metrics are already used in batch job systems. For example, in the Lsbatch batch job system [54], a job can specify its requirements in terms of the number of processors, memory size, disk space size, and the maximum wall clock time it would run. Hence, RUM has an advantage of being time-tested and has a high familiarity with users.

In the context of CMPs, RUM must extend beyond resources that are specified in traditional batch job systems, for example by including the shared cache capacity and off-chip bandwidth rate. In this work, we focus on the shared L2 cache capacity and processor core resources in the QoS target specification. We acknowledge that a complete QoS target would include off-chip bandwidth rate, main memory size, network bandwidth, disk size, and other resources. However, we note that those resources are not specific to CMP design or do not contribute as strongly to the performance variation of a job compared to the L2 cache capacity and processor core. Hence, we leave them for future work.

Optionally, a QoS target may include a *timeslot* resource, which can be specified through a *maximum wall-clock time* which indicates the size of the timeslot, and a *deadline* which indicates the latest expected completion of the timeslot. Maximum wall-clock time is a concept borrowed from batch job systems [54]. It specifies the maximum amount of time that a job should be allowed to run assuming it gets all its requested resources. The maximum wall-clock time is different from WCET in a real-time system in that it does not need to be a safe execution time upper bound. Embedded in it is the users' expectation that a job may be terminated if it runs longer than its maximum wall-clock time. Another reason why timeslot resource specification is optional is that jobs do not necessarily have a maximum wall-clock time or a deadline. Long-running applications, OS daemons, or other legacy applications may not specify timeslot resources, and in this case resources will be allocated to them for their entire lifetime.

In order to make it easier for users to determine an appropriate QoS target, the system may provide several preset RUM targets that users can choose from. Similar preset targets have been employed in many batch job systems. For example, a job can choose one of large, medium, or small configurations. Each configuration comes with preset memory size, maximum processor count, and maximum wall-clock time. However, while preset QoS targets could greatly simplify QoS target selection for users, they may also exacerbate QoS *overspecification*, a situation in which a job needs less resources than what it specifies. This leads to resource fragmentation that will reduce overall throughput. We will address how to recover from resource fragmentation in the following sections.

3.1.3 QoS Execution Modes

Besides QoS target specification, another important component of our QoS framework is how strictly the QoS target must be followed. Similar to the postal delivery system that allows various strictness levels in delivery times, it may be desirable for a utility computing server to provide various strictness levels in meeting the QoS target. To achieve this variety, we propose the following execution modes:

1. **Strict**: The Strict execution mode may be used by jobs that have rigid requirements for a minimum throughput (implied by the RUM) and deadline. To meet a Strict job's QoS target, the requested resources and timeslot must be strictly reserved.

2. **Elastic(X)** [1]: The Elastic(X) execution mode can be used by jobs that have a rigid deadline requirement but can tolerate some deviation of throughput compared to that implied by the amount of resources requested in RUM. The deviation is such that the reduction of throughput (slowdown) is not more than X% compared to the case in which the resources are reserved (i.e. in the Strict mode).

3. **Opportunistic**: The Opportunistic mode may be used by jobs that do not have rigid throughput and deadline requirements. For example, users may use the Opportunistic mode for jobs whose deadlines are still far away.

Note that since we propose to use RUM in our QoS target specification, the Elastic(X) mode is only meaningful when X is defined with a different metric (not a RUM). Otherwise, Elastic(X) mode will be equivalent to the Strict mode with a reduced resource requirement. As a result, for the Elastic(X) execution mode, we specify X in terms of the percentage of slowdown in execution time or cycle-per-instruction (CPI). From users' point of view, the significance of the availability of

[1]The concept of Elastic mode is similar to the one in Buttazzo et al. [5]. However, while in [5] it only applies to periodic jobs, we define the elasticity in terms of slowdown, hence it is applicable to aperiodic jobs as well.

the weaker modes (Elastic(X) and Opportunistic) is that they may match their jobs' requirements better, especially if the weaker modes have lower fee structures. From the system point of view, we will show that weaker modes allow the CMP to boost throughput by accepting more jobs and mitigating QoS target overspecification.

Downgrading QoS Execution Mode. Suppose a user submits a job in Strict mode, and the CMP rejects the job because it cannot meet the QoS target of the job. In this case, the user can probably make the job admissible if he/she is willing to reduce the required resources specified in the QoS target, increase the deadline, or downgrade the mode of their job to a weaker one. We refer to changing the mode from Strict to Elastic(X) or Opportunistic, or from Elastic(X) to Opportunistic by users as *manual mode downgrade*. Since users are fully aware of the consequence of using different execution modes, manual mode downgrade requires the willingness of users to reduce their expectations on the strictness of the jobs' QoS targets.

Mode downgrade can also be performed transparently by the system as long as the old and new modes are *interchangeable*. We define two modes as interchangeable if they can be used to guarantee completion of a job by the same deadline, and throughput variation can be tolerated by the job. Suppose we have a Strict job with deadline of td and maximum wall-clock time of tw, arriving at time ta. We note that there is a time slack of $(td - ta) - tw$. This amount of slack means that the job can be downgraded as an Elastic($\frac{(td-ta)-tw}{tw}$) job while still meeting its deadline. Additionally, it can also be downgraded to the Opportunistic mode for $(td - ta) - tw$

amount of time, but if it has not completed by time $td - tw$, it needs to be switched back to the Strict mode. We refer to these as *automatic mode downgrade.*

3.1.4 The Impact of Execution Mode Downgrade

In this section, we examine the impact of manual and automatic mode downgrade. Before we continue the discussion, it is helpful to distinguish two factors that contribute to sub-optimal throughput when there are only Strict jobs. The first is *external resource fragmentation* which refers to idle resources that are not allocated to any jobs. Typically, external resource fragmentation occurs when there are not sufficient remaining available resources to accept a new job. The second factor is *internal resource fragmentation* which is caused by a job not using all resources that are allocated to it.

First, let us consider the impact of manual mode downgrade as illustrated in Figure 3.2. In the figure, each bar represents the time each job takes to complete its computation, the x-axis represents the time duration and the y-axis represents different accepted jobs. In this example, we assume that jobs are sequentially submitted to the system, and the first six accepted jobs are shown in the figure. Assume that each job requires 40% of the shared cache size in order to complete in T time. The deadline of each job is assumed to be 1.5T from the time when the job is accepted. If there are six Strict jobs (Figure 3.2(a)), at most two jobs can be executed simultaneously since there is not enough cache space to run more than two jobs simultaneously. The

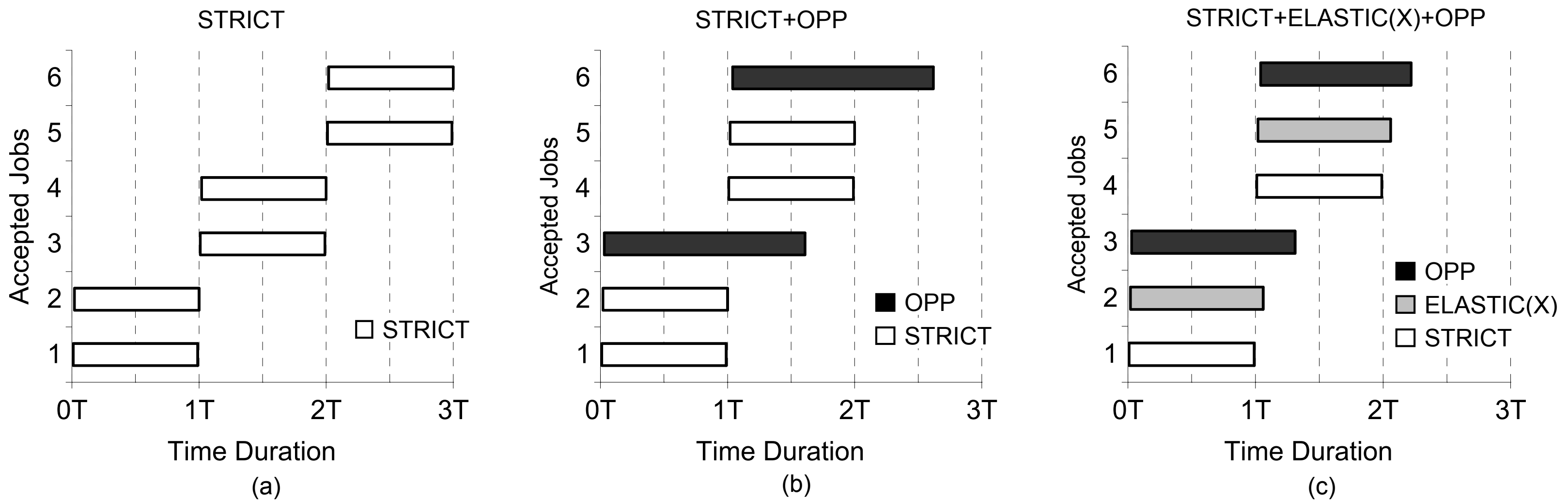


Figure 3.2: Illustration of the impact of manual mode downgrade: all Strict jobs (a), third and sixth accepted jobs are manually downgraded to Opportunistic (b), and second and fifth accepted jobs are downgraded to Elastic(X) (c).

external resource fragmentation includes two idle cores in a 4-core CMP and 20% of the L2 cache capacity. The CMP takes 3T to finish all six jobs and all of them meet their deadlines. However, if users manually downgrade the third and sixth jobs to Opportunistic jobs (Figure 3.2(b)), the system can accept and run more jobs simultaneously and reduce external resource fragmentation. Although the third and the sixth jobs run slower, the overall throughput is improved as it only takes slightly more than 2.5T to complete all six jobs. If users also manually downgrade the second and fifth jobs to Elastic(X) jobs, the system can employ the resource stealing technique (Section 3.2) to discover unused cache capacity allocated to Elastic(X) jobs and reallocate them to Opportunistic jobs. This results in the third and sixth jobs completing faster, while the second and fifth jobs run slower but still meet their deadlines. The overall throughput is potentially improved further as we also reduce internal cache fragmentation.

With manual mode downgrade, when a Strict job is downgraded to Elastic(X), its unused resources are re-allocated to Opportunistic jobs. However, since the job may be slowed down by up to X%, in order to guarantee meeting its deadline, the job needs to reserve resources for a longer time duration of $tw \times (1 + X)$ (versus tw if it remains a Strict job). Since the same amount of resources are reserved for a longer time, the ability of the CMP to accept future jobs may be reduced, which in turn may reduce future throughput. Consequently, manual downgrade of a Strict job to Elastic(X) may reduce throughput if it is not accompanied by a throughput increase due to Opportunistic jobs benefiting from reallocation of excess resources. Overall, we can expect throughput improvement to be higher when there are both Elastic(X) and Opportunistic jobs to complement Strict jobs, compared to when there are only Opportunistic jobs to complement Strict jobs.

With automatic mode downgrade, an additional impact occurs when a Strict job is downgraded to Opportunistic mode. In contrast to the manual mode downgrade in which an Opportunistic job does not reserve any resources, with automatic mode downgrade the resources requested by the job still need to be reserved for the length of its maximum wall-clock time. The job can only be run in Opportunistic mode before it meets its reserved timeslot, by which time it has to switch back to Strict mode in order to ensure that its deadline is met. When a job completes before it meets its reserved timeslot, the reserved resources can be reclaimed to allow new jobs to be accepted and future throughput to be improved. As a result, the reserved timeslot needs to be placed as far away as possible in order to increase the probability that the job completes before the reserved timeslot is encountered. Finally, we do not consider automatically downgrading a Strict job to Elastic(X) job because the Elastic(X) mode reserves resources longer than the original Strict mode, which is likely to be detrimental to throughput.

Comparing manual and automatic mode downgrade, if the manually downgraded Opportunistic jobs do not suffer from too much slowdown due to lack of resources, it can be expected that manual downgrade would achieve a higher throughput than automatic downgrade since the latter still relies on resource reservation, which may reduce job admission rate. However, the automatic mode downgrade is still useful because it does not rely on users' willingness to downgrade their jobs to weaker modes.

3.2 Resource Stealing

In this section, we will discuss resource stealing, a key technique that supports the proposed Elastic(X) execution mode.

3.2.1 Managing Cache Capacity Partitions

In order to track and control the shared cache allocation across cores, we need to employ a cache partitioning scheme. Cache partitioning can be achieved through a global approach or per-set approach. In the global approach, a modified LRU policy [48] keeps a global counter that tracks the number of cache blocks currently allocated to each core, and another counter that records the target number of cache blocks that should be allocated to each core. On a cache miss, the victim block is chosen from the blocks that belong to the core which has more allocated blocks than its target number of blocks. This process is repeated until each core reaches its target cache allocation. The number of blocks allocated to a core in different sets varies, but the sum of them over all sets would match the target allocation. We note that while this global approach is relatively simple to implement, it has a drawback that the distribution of blocks allocated to an application in different cache sets varies across different runs as it is affected by other applications that run simultaneously. This variation of allocation, especially when it occurs in heavily-used sets sometimes introduces a large variation in miss rates and performance for the same application across different runs. As a result, we do not use it in our QoS framework.

The per-set cache partitioning algorithm in [21, 33] is a finer-grain version of the modified LRU replacement policy proposed by Suh et al. [48]. In [21, 33], each core is assigned a target allocation counter that records the number of *cache ways* that should be assigned to it. Each set in the cache also has per-set counter to track the number of blocks in the set that are currently allocated to the core. When a core suffers a cache miss on a set, the core's per-set counter is compared against its target allocation counter. If the per-set counter has a lower value, a block that belongs to one of the over-allocated cores is selected as the victim. Otherwise, a block that belongs to the core itself is selected as the victim. Over time, the number of blocks allocated to a core will be the same over all sets. Consequently, over different runs, the same job with the same cache space allocation will perform more uniformly. This is a desirable factor in a system that tries to provide QoS.

Our cache partitioning scheme is based on the fine-grain approach in [21, 33], but we adapt it to our QoS framework. In our modified version, the selection of the victim block for a cache miss also depends on the execution mode of the job that the victim belongs to. On each cache miss, if there is more than one over-allocated core, the victim is first selected from an over-allocated Strict or Elastic(X) job (if there is any). Otherwise, the LRU block among the blocks from Opportunistic jobs is selected as the victim. The reason why over-allocated Strict/Elastic(X) jobs are given a higher priority for victim selection is that we would like to accelerate the cores running these jobs in converging to their target allocations, and reallocate the excess cache capacity from Elastic(X) jobs to Opportunistic jobs as fast as possible.

3.2.2 Criterion for Resource Stealing

In Section 3.1.3, the Elastic(X) execution mode was proposed in order to enable the system to remove excess cache capacity allocated to a job due to QoS ovespecification, and X specifies the maximum slowdown (*slack*) that is acceptable to the user. When applying resource stealing, it is difficult to accurately measure how much CPI increases when we employ resource stealing versus when we do not. Hence, we need a more measurable metric.

First, we note that the components of CPI are additive, i.e. the overall CPI is the sum of the CPI assuming an infinite cache and the additional CPI when cache misses are considered [12, 30]. Specifically, assuming a system with two levels of caches, the overall CPI of an application can be expressed as:

$$CPI \quad = \quad CPI_{L2->\infty} + h_m \times t_m \tag{3.1}$$

where $CPI_{L2->\infty}$ indicates the CPI of the program when the lowest level on-chip cache (e.g. the L2 cache) has infinite size, h_m and t_m indicate the number of L2 misses per instruction and the penalty (in number of cycles) of an L2 miss respectively. With resource stealing, we attempt to steal L2 cache capacity from an Elastic(X) job and reallocate it to Opportunistic jobs. The effect of reduced L2 cache size for the Elastic(X) job is a higher L2 cache miss rate (hence higher h_m), but other variables would remain roughly unchanged [2]. Since $h_m \times t_m$ is only one component of the CPI, and all other components have positive (at least non-negative) values, an increase of X% in h_m would result in a less than X% increase in CPI. We exploit this observation to guide our resource stealing algorithm such that it removes cache capacity from an Elastic(X) job but without increasing the job's L2 cache miss rate by more than X%.

Note that our criterion of allowing the L2 cache miss rate to increase by not more than X% is likely to be conservative, i.e. the increase in CPI is smaller than X%. We will address this issue in Section 3.3. Finally, monitoring the L2 miss rate increase is achievable with relatively simple hardware modification which will be described in the next section. Therefore, in this dissertation, we choose to use L2 miss rate as the metric to guide the resource stealing mechanism.

3.2.3 Microarchitecture Support

In order to monitor the miss rate increase due to partition changes, we need a mechanism to dynamically obtain the miss rates for both the reduced and original partition cases. To achieve this, we use a straightforward method that utilizes an additional *duplicate cache tag array* [35, 47] that keeps track of what blocks the cache would have if resource stealing had not been applied, while the main cache tag array

[2] The CPI of infinite L2 cache ($CPI_{L2->\infty}$) is mostly affected by physical L1 and L2 cache organization, and not affected much by L2 cache partition sizes. The latency of an L2 miss (t_m) may be affected as memory bus contention increases due to the increase in the number of L2 cache misses. However, this can be mitigated by prioritizing memory requests from Elastic(X) jobs over those from Opportunistic jobs. In addition, we can monitor bus utilization and disable resource stealing when bus saturation is reached. According to Little's Law [18], prior to saturation, queueing delay is roughly constant.

keeps track of the actual cache content. To reduce the storage overhead of keeping the duplicate tag array, we employ set sampling [35, 36], in which only a few sets are augmented with duplicate tags and profiled to infer the global cache behavior. We let the stream of all L2 cache accesses be visible to both tag arrays so that only their numbers of misses differ.

When an Elastic(X) job runs, the resource stealing algorithm is activated. The algorithm reduces the Elastic(X) job's partition size by one way. This "stolen" way is re-allocated to one of the Opportunistic jobs. The target allocation counters for the Elastic(X) job and the Opportunistic job are updated, which allows the cache to converge to the new partitions over time. In the meantime, the duplicate tags keep track of the total number of misses for the Elastic(X) job without resource stealing. If the extra number of misses in the main tags reaches or exceeds X% compared to that in the duplicate tags, then the resource stealing has potentially caused the Elastic(X) job to slowdown by more than X%, so the resource stealing is canceled and all the stolen ways are returned to the Elastic(X) job. Otherwise, in the next time interval we steal another way from the Elastic(X) job. Note that while repartitioning occurs at periodic intervals, the number of misses in the duplicate tag array and main tag array are not reset at each interval. This ensures that the *total* number of misses since the start of the Elastic(X) job does not increase by more than X% due to resource stealing.

3.3 More Effective Resource Stealing Techniques

The resource stealing algorithm presented in Section 3.2, which we refer to as RS_{base}, is fairly simple, but the criterion of allowing the L2 cache miss rate to increase by not more than X is too conservative. This section presents the reason why the RS_{base} algorithm is conservative in Section 3.3.1, and two improved resource stealing techniques in Section 3.3.2 and Section 3.3.3. Finally, a mechanism to dynamically control the resource stealing is presented in Section 3.3.4.

3.3.1 The Conservativeness of the RS_{base} Algorithm

Equation 3.1 expresses the CPI as addition of its components, with one of the components being $h_m \times t_m$. Let CPI and h_m denote the original CPI and the original number of L2 misses per instruction, i.e. when resource stealing is not applied. Let CPI' and h'_m denote the CPI and the number of L2 misses per instruction when resource stealing is applied. Since both $CPI_{L2->\infty}$ and t_m can be assumed to be unaffected by resource stealing, the increase in CPI, denoted as $\triangle CPI$, can be expressed in Equation 3.2. Consequently, the miss rate increase, denoted as $\triangle MissRate$, can be expressed in Equation 3.3.

$$\triangle CPI = \frac{CPI' - CPI}{CPI} = \frac{(h'_m - h_m) \times t_m}{CPI} \tag{3.2}$$

$$\triangle MissRate = \frac{h'_m - h_m}{h_m} = \triangle CPI \times \frac{CPI}{h_m \times t_m} \tag{3.3}$$

Equation 3.3 indicates that to satisfy the condition that CPI does not increase by more than X, the maximum increase in miss rate should be X multiplied by a factor, denoted by $\mathcal{F}$, which is $\frac{CPI}{h_m \times t_m}$. $\mathcal{F}$ is always equal or larger than one since $h_m \times t_m$ is only one component of CPI. The RS_{base} algorithm conservatively assumes that $\mathcal{F}$ is one. While safe, the algorithm is not effective because in practice, $\mathcal{F}$ is often much larger than one. Figure 3.3 shows $\mathcal{F}$'s of fifteen SPEC2006 benchmarks

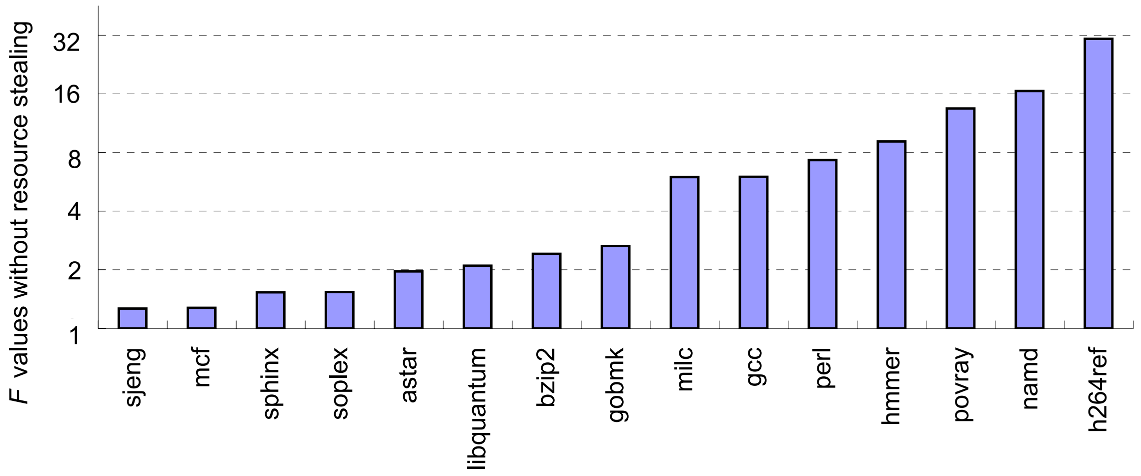


Figure 3.3: The $\mathcal{F}$ (i.e. $\frac{CPI}{h_m \times t_m}$) values of fifteen SPEC2006 benchmarks. Detailed simulation parameters are described in Section 3.5.

without applying resource stealing. While $\mathcal{F}$ varies between 1.26 and 30.1 for different benchmarks, for two thirds of the benchmarks the value is larger than 2, indicating that to tolerate a slowdown of X, the miss rate can be allowed to increase by more than 2X. Therefore, assuming $\mathcal{F}$ to be one as in RS_{base} is too conservative.

3.3.2 CPI Sampling

Our first attempt to make resource stealing more effective is by estimating the value of $\mathcal{F}$ to get the maximum allowable increase in L2 cache miss rate. Since h_m can be obtained on-line through the duplicate cache tag array and t_m is unaffected by resource stealing, the challenge is to obtain CPI, which is the actual CPI of the program when no resource stealing is applied. Unfortunately, once resource stealing is applied, the CPI is affected and its original value is no longer obtainable. To approximate the value, we rely on CPI sampling in order to predict its original value. More specifically, an application's execution is divided into intervals with equal number of dynamic instructions. We choose one interval for sampling CPI (no resource stealing is applied), followed by α number consecutive intervals in which resource stealing is applied, where α is an adjustable parameter. During a sampling interval, we cancel all resource stealing and use the original cache partition size in the main tag array. The CPI value collected in each sampling interval forms a sample, and we assume that these samples are independent and identically distributed. The average value of sampled CPIs ($\overline{CPI}$) form a point estimation of the original CPI value, and we can use it for computing $\mathcal{F}$. In addition, since during sampling no resource stealing is applied, its CPI is not increased. Therefore, the amount of CPI increase during resource stealing can be adjusted up by a factor of $\frac{\alpha+1}{\alpha}$. Hence, the maximum allowable increase in L2 cache miss rate can be expressed as:

$$\triangle MissRate \leq X \times \frac{\overline{CPI}}{h_m \times t_m} \times \frac{\alpha + 1}{\alpha} \tag{3.4}$$

where X is the maximum allowable slowdown. We refer to this resource stealing algorithm as RS_{sample}. The major risk of using the RS_{sample} algorithm is the accuracy of estimating the original CPI with the average of sampled CPIs ($\overline{CPI}$). If the $\overline{CPI}$ overestimates the original CPI, the miss rate may increase beyond a safety threshold and the application slowdown may exceed X. An example in which this can happen is when the CPI samples capture unusually high CPI which cause the accumulated $\overline{CPI}$ to permanently overestimate the original CPI. In general, the larger the fraction of execution time dedicated as sampling intervals, the more accurate the estimation accuracy becomes. However, it also reduces resource stealing opportunities and effectiveness. In the next section, we add margin of safety to the CPI estimation in order to avoid CPI estimation error to cause the application's slowdown to exceed X.

One intuitive question regarding the use of CPI sampling is whether we can directly use it to guide the increase in CPI without regarding the miss rate increase. For example, we can compare the average CPI when resource stealing is applied versus the average CPI during sampling in which resource stealing is not applied. Resource stealing can be applied when the difference between the two average CPIs is still smaller than X. While such a technique is clearly possible, we note that the risk is high that the slowdown may overshoot X and it is very difficult to detect and recover from it. In contrast, using miss rate increase as a proxy allows the technique to switch seamlessly between more aggressive resource stealing techniques with the safer technique of RS_{base}. For example, if for some reasons we suspect the slowdown overshoots X, resource stealing can be canceled and we can wait until the cumulative miss rate increase is less than X before attempting more aggressive resource stealing again.

3.3.3 Adding Safety Margin

In order to add a safety margin to the aggressive resource stealing RS_{sample} that we introduced earlier, we utilize the *degree of confidence* of the estimation. One question is whether the degree of confidence should be computed for all CPI samples or for most recent ones. We note that applications are known to go through phases of execution in which one phase often exhibits a very different CPI compared to prior phases. Hence, the assumption that sampled CPIs are identically distributed are more valid over a short time period than over a long time period. Hence, we define a *sample safe size*, denoted by W, as the number of the most recent collected samples considered in computing the degree of confidence. We assume that CPI samples over W are independent and identically distributed with a mean of μ and variance of σ^2. Consequently, according to Central Limit Theorem, $\overline{CPI}$ over W follows a normal distribution with a mean of μ and variance of $\sigma^2_{sample} = \frac{\sigma^2}{W}$. Hence, there is a 95% probability that the original CPI is larger than $\overline{CPI} - 1.64 \times \sigma_{sample}$.

Similarly, there is a 99% probability that it is larger than $\overline{CPI} - 2.33 \times \sigma_{sample}$. Hence, $\overline{CPI} - 1.64 \times \sigma_{sample}$ forms the lower bound of the original CPI value with 95% confidence. We denote this estimated lower bound as CPI_{min}, and our criterion for miss rate increase becomes:

$$\triangle MissRate \leq X \times \frac{CPI_{min}}{h_m \times t_m} \times \frac{\alpha + 1}{\alpha} \tag{3.5}$$

We refer to this modified algorithm as RS_{safe}. Compared with RS_{sample}, RS_{safe} is relatively conservative in stealing resource since CPI_{min} includes a margin of safety that is often lower than the actual CPI. During a period in which CPI samples are unusually volatile (i.e. their values highly fluctuate), the estimated standard deviation becomes high, CPI_{min} becomes small, hence miss rates are not allowed to increased by much. However, when CPI samples are relatively stable, the estimated standard deviation becomes low, CPI_{min} becomes large, and miss rates are allowed to increase significantly. Hence, the algorithm tends to be more aggressive when stable phases are detected, and becomes conservative when unstable phases or transitions between phases are encountered. In addition, a temporary large spike in CPI values does not cause overshoot in slowdown. The reason is that while the average CPI over the samples becomes elevated, the CPI_{min} does not change much because the estimated standard deviation also increases significantly. In some of these cases, we actually observe that the computed $\mathcal{F}$ drops below one, which is unnecessarily low. In this case, we simply replace it by one since we know it is safe to do so, i.e. we let the algorithm to temporarily revert back to RS_{base}.

The sample safe size W obviously impacts the effectiveness of the RS_{safe} algorithm. Ideally, W must be adjusted such that it is able to detect most of program phases with stable CPI values. If W is too large, it may include multiple phases with highly different CPI values, and the algorithm becomes overly conservative and has less opportunity to perform resource stealing. However, if W is too small, the accuracy of average and variance of sampled CPIs is reduced. In our experiments, we found a sample safe size of three to deliver reasonable accuracy and coverage for resource stealing.

3.3.4 Dynamic Resource Stealing Mechanism

So far we have discussed two alternative techniques to more effectively steal cache capacity from Elastic(X) jobs. However, we note that the resource stealing techniques alone are not sufficient to ensure improving the *overall* throughput. This is because as mentioned in Section 3.1.4, the overall throughput can be improved only when the Elastic(X) jobs are accompanied by the opportunistic jobs which benefit from the relocation of the excess cache capacity. Otherwise, the overall throughput may be even reduced due to the slowdown of the Elastic(X) jobs. Ideally, in order to improve overall throughput, resource stealing should be suspended temporally when the opportunistic jobs cannot benefit from extra cache space and be resumed when it can help reduce the misses of opportunistic jobs significantly. In this section, we present a mechanism, which dynamically controls the suspension/resuming of the resource stealing by counting the reduction of total number of cache misses. We refer to this mechanism as RS_{dyn}. It works as follows.

Let C_{RS} and C_{NoRS} denote the total number of cache misses with and without resource stealing respectively and they are set to 0 when resource stealing is activated. After each cache repartitioning interval expires, following steps will be performed:

1. The total number of cache misses in previous interval obtained from the main tags is added to C_{RS} if resource stealing is not suspended, otherwise it is added to C_{NoRS}. The total number of cache misses obtained from the duplicate tags is added to C_{RS} if resource stealing is not suspended, otherwise it is added to C_{NoRS}.

2. C_{RS} and C_{NoRS} are compared to determine whether the resource stealing is useful or not. In this dissertation, we use X to guide the suspension/resuming of the resource stealing such that: resource stealing will be suspended (if it has not) when $\frac{C_{RS}}{C_{NoRS}} > 1 - X$, and will be resumed (if it has not) when $\frac{C_{RS}}{C_{NoRS}} \leq 1 - X$. The intuition of using this comparison is that the total number of cache misses should at least be reduced by X under the cost of the slowdown of Elastic(X) jobs.

3. Upon the **suspension**/*resuming* of the resource stealing, we exchange the values of the allocation counters of the main tags and duplicate tags for all processors and apply the resource stealing on the **duplicate**/*main* tags. i.e., in the

next repartitioning interval, the partition sizes of **duplicate**/*main* tags will be changed according to the running resource stealing algorithms and those of the **main**/*duplicate* tags will keep unchanged. As a result, we are able to estimate the total number of misses with and without resource stealing through C_{RS} and C_{NoRS} respectively no matter resource stealing is suspended or not.

Note that RS_{dyn} slightly affects on the running resource stealing algorithms. The number of cache misses of Elastic(X) job **with**/*without* resource stealing should be accumulated from **main**/*duplicate* tags when resource stealing is not suspended and from **duplicate**/*main* tags if it is. The CPI sampling technique used in RS_{sample} and RS_{safe} is not affected by RS_{dyn}. In addition, C_{RS} and C_{NoRS} may not accurately estimate the total number of cache misses with and without resource stealing respectively since the contents in main tags and duplicate tags are not consistent upon the suspension or resuming of resource stealing. However, the impact of such inaccuracy can be largely mitigated by increasing the size of repartitioning interval.

3.4 Local Admission Controller Implementation

For the Local Admission Controller (LAC), we implement a First Come, First Served (FCFS) scheduling algorithm using a basic resource allocation model from [38]. The LAC maintains a list of vectors that encode processor core and cache capacity resources and the timeslots in which they are available. A job specifies its QoS target through a resource request vector which encodes the amount of resources needed for each resource type, and for how long (based on the maximum wall-clock time) they are needed. For a Strict or Elastic(X) job, The LAC tries to find the *earliest* timeslot in which this vector can be fit before the job's deadline. If such a timeslot is found, the job is accepted and the resources are reserved in that timeslot. An Opportunistic job is always accepted if there are spare resources not already taken up by Strict or Elastic(X) jobs.

To keep the OS thread scheduler unchanged, we implement the LAC as a user-level program. The LAC has a scheduler queue to store accepted jobs and manage their resource and timeslot reservations. Once it is time to start a job, the job is submitted to the OS. If the job is an Elastic(X) job, resource stealing is also activated. To avoid timesharing from violating a Strict or Elastic(X) job's deadline, the LAC pins only one such job to one processor core. However, the LAC may pin multiple Opportunistic jobs on a core that is not already assigned to Strict or Elastic(X) jobs.

3.5 Evaluation Methodology

Simulation Environment. To evaluate the proposed QoS framework, we use a full-system simulator based on Simics [31] to model a 4-core CMP node with Fedora Core 4 Linux as the operating system. Each core is an in-order processor with a 2GHz clock frequency. Each core has a private L1 instruction and a private L1 data cache with a 32KB size, 4-way associativity, 64-byte block size, LRU replacement policy, write back policy, and 2-cycle access time. The L2 cache is shared by all four cores, and is a unified cache with a 2MB size, 16-way associativity, 64-byte block size, modified LRU replacement policy (Section 3.2.1), write-back policy, and 10-cycle access time. The main memory is 4GB in size with a 300-cycle access time. The peak bandwidth to the main memory is 6.4GB/s. The simulator ignores the impact of page mapping by assuming each job is allocated contiguous physical memory.

Resource Stealing. We employ set sampling to implement the duplicate tags used in resource stealing. The duplicate tags only cover $\frac{1}{8}$ of the total number of sets, which is conservative compared to 32 sets used in [36]. Every 8^{th} set is sampled. The interval for triggering cache repartitioning is 2 million instructions of the Elastic(X) job.

To evaluate the RS_{sample} and RS_{safe} algorithms, the sampling parameter α is set to two for both RS_{sample} and RS_{safe} algorithms, which means that 33% of the overall intervals are spent on sampling and the rest for resource stealing. α is chosen to be small in order to achieve a high estimation accuracy. In each sampling interval, the first one million instructions are excluded from sampling as the cache is adjusting to the new partition size. The CPI of the next one million instructions is used to form a sample point. For the RS_{safe} algorithm, the degree of confidence is 95% is used, and the sample safe size of three is used.

Individual Job. To evaluate our proposed QoS execution modes and resource stealing technique, we choose fifteen C/C++ benchmarks from SPEC2006 benchmark suite [45]: *gcc, bzip2, perl, gobmk, mcf, hmmer, sjeng, libquantum, h264ref, milc, astar, namd, soplex, povray, sphinx*. We use the *ref* input sets for all benchmarks except for *milc* and *soplex* which use the *train* input sets because their ref input sets require too much memory. For each benchmark, we first inspect its source code and identify its initialization routines. Then we skip the initialization routines and simulate the next 200 million instructions.

To reduce the number of benchmarks that we need to evaluate, we classify the benchmarks according to their cache space sensitivity. For each benchmark, we calculate the CPI increase when we reduce its L2 cache allocation from 7 ways to 1 way, and from 7 ways to 4 ways. Then we plot them in a two-dimensional space (Figure 3.4).

From the figure, we can roughly classify the fifteen benchmarks into three groups based on how sensitive they are to the allocated cache space: *highly sensitive* (Group 1), *moderately sensitive* (Group 2), and *insensitive* (Group 3). Similar classification methods can be found in [8, 36]. Highly sensitive benchmarks are ideal beneficiaries

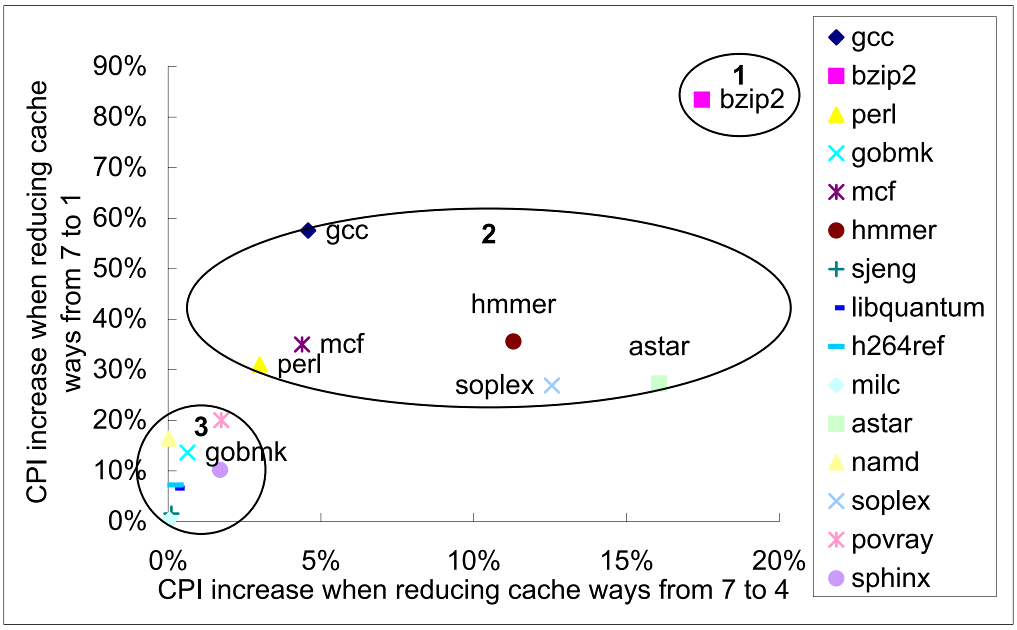


Figure 3.4: The sensitivity of each benchmark to cache capacity.

of resource stealing, whereas insensitive benchmarks are ideal donors for resource stealing. From each group, we choose one representative benchmark: *bzip2* from Group 1, *hmmer* from Group 2 and *gobmk* from Group 3. The L2 miss rates and L2 misses per instruction of these benchmarks when they are allocated 7 cache ways are listed in Table 3.1.

Table 3.1: The benchmarks used as individual jobs in the evaluation.

Benchmark	Input Set	L2 Miss Rate	L2 Misses Per Instruction	Number of Skipped Instructions
bzip2	ref.chicken	20%	0.0055	315M
hmmer	ref.retro	17%	0.001	0.3M
gobmk	ref.nngs	24%	0.004	267M

Workload Composition. We construct 10-job workloads and measure the wall clock time to complete all ten jobs. Each job requests a processor core and L2 cache capacity of 896KB (7 ways in the 16-way L2 cache). We assume incoming jobs with Poisson arrival, with an inter-arrival time that assumes full computation capacity utilization of a 128-CMP server. Specifically, on a 4-core CMP, in one job's wall-clock time, there are on average 4×128 new jobs that arrive and probe the CMP's Local Admission Controller. Job deadlines are assigned as follows. We pseudo-randomly set 50% of them with a *tight deadline* ($td - ta = 1.05 \times tw$), 30% with a *moderate deadline* ($td - ta = 2 \times tw$) and 20% with a *relaxed deadline* ($td - ta = 3 \times tw$).

To evaluate the various execution modes, we use five configurations shown in Table 3.2. The base configuration is *All-Strict*. The *EqualPart* configuration mimics the Virtual Private Cache [33] by equally partitioning the cache capacity among all cores, but without an admission controller and bandwidth partitioning. In the *Hybrid-2* configuration, unless otherwise specified, we employ RS_{safe} algorithm and a slack value of 5% for each Elastic(X) job. The detailed evaluation of different resource stealing techniques is presented in Section 3.6.

The 10-job workloads are constructed in two ways. First, we use instances of the same benchmark, i.e. 10 instances of *bzip2*, 10 instances of *hmmer*, and 10 instances of *gobmk*. In addition, we also construct two mixed-benchmark workloads shown in Table 3.3. Mix-1 is a favorable workload for our framework since cache-sensitive *bzip2* is the recipient of resource stealing, while cache-insensitive *gobmk* is the donor

Table 3.2: Execution modes configurations.

Configuration	Percentage of Jobs in Various Execution Modes
All-Strict	100% Strict
Hybrid-1	70% Strict + 30% Opportunistic
Hybrid-2	40% Strict + 30% Elastic(5%) + 30% Opportunistic
All-Strict+ AutoDown	100% Strict, jobs with moderate or relaxed deadlines are automatically downgraded.
EqualPart	No admission control, default Linux job scheduling, L2 cache is equally partitioned among cores.

of resource stealing. In contrast, Mix-2 is not favorable to our resource stealing technique.

Table 3.3: Mixed-Benchmark Workloads.

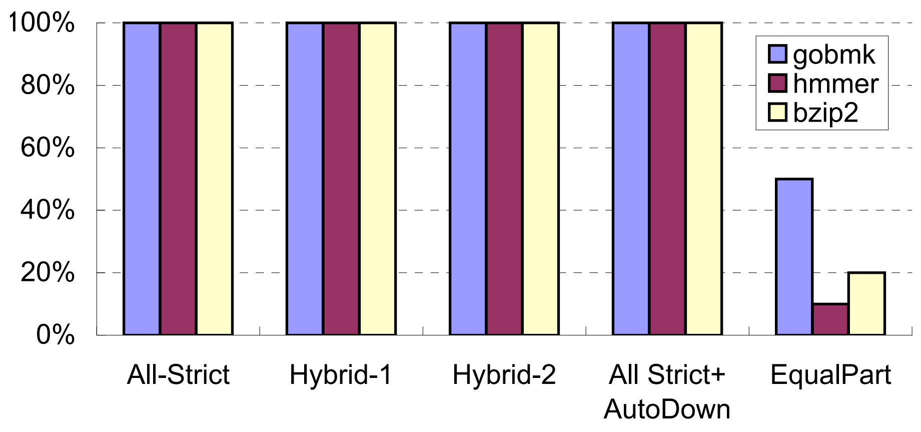
Type	Workload Composition
Mix-1	*hmmer* (Strict), *gobmk* (Elastic(5%)) and *bzip2* (Opportunistic)
Mix-2	*hmmer* (Strict), *bzip2* (Elastic(5%)) and *gobmk* (Opportunistic)

3.6 Evaluation Results

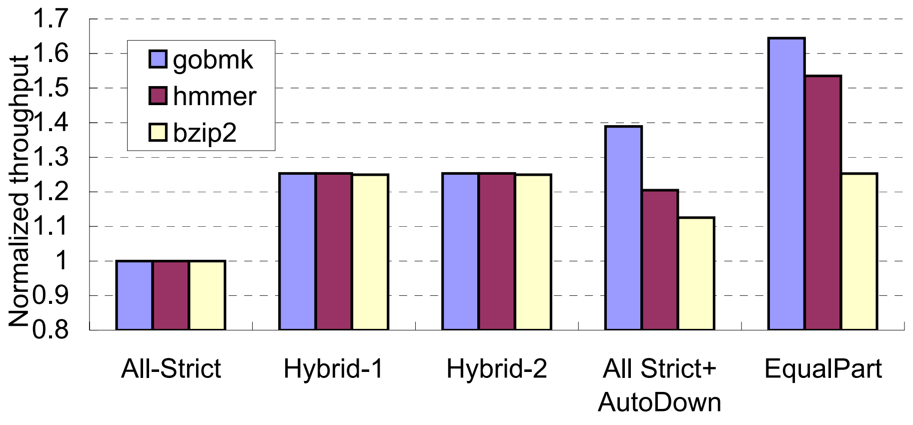
In this section, we will present our experiment results, including the impact of various execution modes (Section 3.6.1), the impact of automatic mode downgrade (Section 3.6.2), evaluation of mixed workloads (Section 3.6.3), evaluation of the resource stealing techniques (Section 3.6.4), the impact of slack X and RS_{dyn} mechanism on overall throughput (Section 3.6.5) and characterization of the Local Admission Controller (Section 3.6.6).

3.6.1 Impact of Various Execution Modes

Let *deadline hit rate* refer to the fraction of jobs that meet their deadlines. Figure 3.5(a) shows the deadline hit rates for various configurations (Table 3.2) with workloads consisting of ten identical instances of a single benchmark. For our QoS framework, the deadline hit rate is only computed for Strict and Elastic(X) jobs. The figure shows a consistent result of 100% deadline hit rate in our QoS framework. In contrast, the deadline hit rates are only 50%, 10% and 20% in EqualPart for *gobmk*, *hmmer* and *bzip2*, respectively. This is because in EqualPart, the lack of an admission controller causes jobs to be accepted continuously despite the fact that the CMP no longer has sufficient computation capacity to meet the jobs' deadlines. This observation reinforces the argument that partitioning the cache capacity among cores alone cannot fully provide QoS. Only after incorporating an admission control policy and using appropriate QoS targets can jobs reliably meet their deadlines.



(a) Deadline hit rates for Strict and Elastic(X) job



(b) Job throughput

Figure 3.5: Comparing QoS and throughput of different configurations.

Figure 3.5(b) compares the job throughput of the single-benchmark workloads on various configurations, measured as the total wall-clock time to complete the first ten accepted jobs. The throughput is normalized to the All-Strict case. Comparing All-Strict and EqualPart, the figure clearly shows that in all workloads, providing strict QoS comes at a cost of significantly lower job throughput, e.g., the throughput in EqualPart is higher by 64%, 54% and 25% for *gobmk*, *hmmer* and *bzip2*, respectively. There are several factors that cause this result. The first is various external resource fragmentation such as the processor cores (only two jobs run simultaneously, leaving two idle cores) and cache capacity (only 14 ways in the 16-way L2 cache are allocated). The second is the internal cache capacity fragmentation if the jobs do not fully utilize their allocated cache partitions, which is especially the case for cache-insensitive benchmarks such as *gobmk*. In contrast, the EqualPart configuration does not suffer from any external resource fragmentation and suffers little from internal resource fragmentation. As a result, the more cache sensitive the jobs in a workload are, the smaller throughput reduction the workload suffers from when QoS is provided.

In Hybrid-1, the existence of Opportunistic jobs effectively removes external core and cache capacity fragmentation, resulting in a significant 25% improvement in throughput for all workloads. However, since internal cache capacity fragmentation still exists (especially in *gobmk* and *hmmer*), the throughput is still lower than that in EqualPart. In Hybrid-2, the existence of Elastic(X) jobs conceptually allows some cache capacity to be stolen and reallocated to Opportunistic jobs, reducing some internal cache capacity fragmentation. However, the figure shows that the throughputs of all the workloads in Hybrid-2 are almost the same as those in Hybrid-1. Upon closer analysis, we find that indeed Opportunistic jobs complete sooner but the overall job throughput is largely unaffected because the tenth accepted job is a Strict job and it completes at almost the same time in Hybrid-1 and Hybrid-2 for all workloads. Note that internal cache capacity fragmentation still exists in Strict jobs (70% in Hybrid-1 and 40% in Hybrid-2) and it cannot be removed without the risk of reducing throughput and missing deadlines. This prevents Hybrid-1 and Hybrid-2 from matching the throughput of EqualPart.

Finally, in All-Strict+AutoDown, automatic mode downgrade is applied to Strict jobs to remove some of the resource fragmentation, resulting in throughput improvements of 39%, 20% and 13% for *gobmk*, *hmmer* and *bzip2* respectively. Again, due to their higher internal cache capacity fragmentation, the throughput improvement for *gobmk* and *hmmer* are higher than that of *bzip2*. The throughput improvement over All-Strict is substantial, especially considering that the optimization does not require users to downgrade the modes of their jobs. However, we note that since we apply automatic mode downgrade only to jobs whose deadlines are relaxed or moderate (Table 3.2), only half of the jobs benefit from automatic mode downgrade. We can expect that if more jobs have moderate or relaxed deadlines, the throughput will be closer to that of EqualPart.

Overall, we conclude that while providing QoS imposes a severe penalty on throughput, a significant part of the throughput can be recovered through several schemes: providing various execution modes to users, and/or employing automatic mode downgrade transparently.

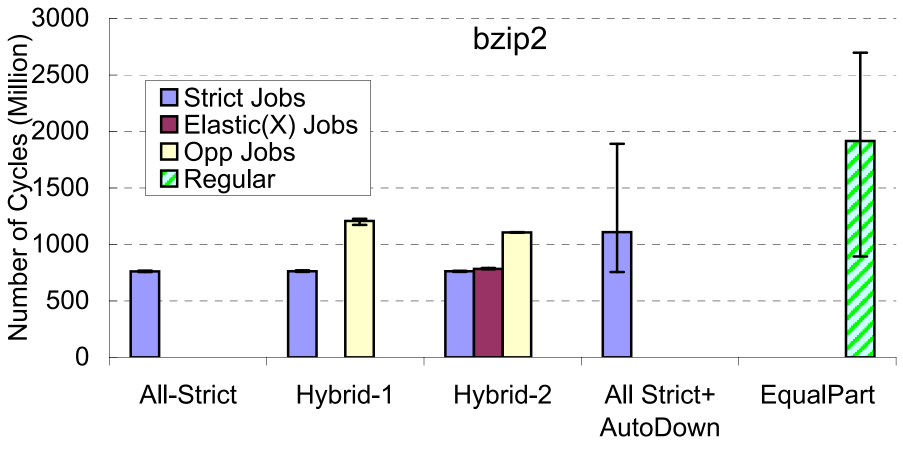
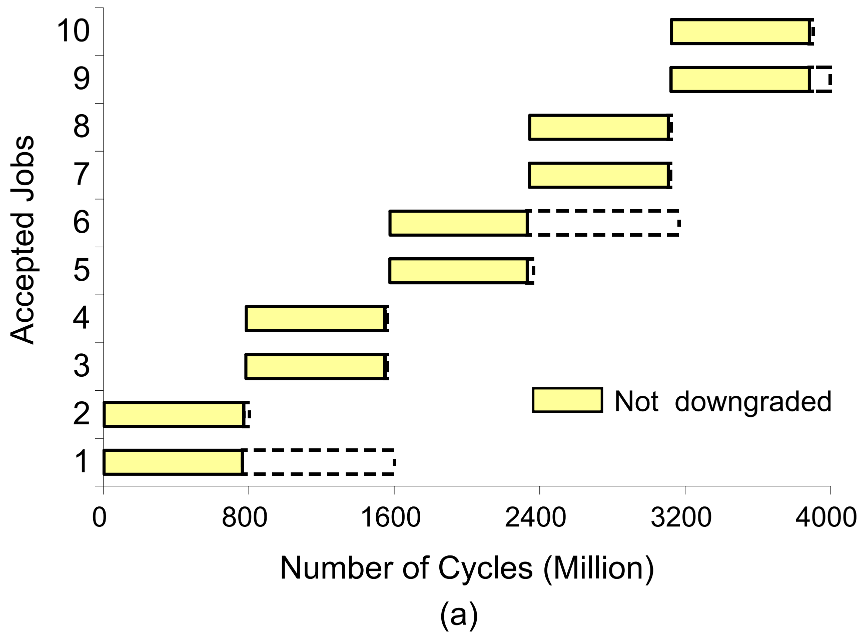


Figure 3.6: The average and variation of wall-clock time for different configurations. The jobs are instances of *bzip2*.

To further analyze the impact of various execution modes on wall-clock time, Figure 3.6 shows the average wall-clock time of jobs in different configurations for the single-benchmark workload of *bzip2*. The candle on each bar shows the range between the minimum and the maximum wall-clock time. The figure shows that with our QoS framework, Strict jobs in all configurations except All Strict+AutoDown have short and almost-constant wall-clock times. The Elastic(X) jobs in Hybrid-2 run slightly longer than Strict jobs because of resource stealing. However, their wall-clock time is still absent of much variation. As expected, because resources are not reserved, Opportunistic jobs in Hybrid-1 and Hybrid-2 have a higher average and variation of wall-clock time compared to Strict jobs. Opportunistic jobs in Hybrid-2 have lower wall-clock time compared to those in Hybrid-1, thanks to the extra cache capacity stolen from Elastic(X) jobs. We can see the impact of automatic mode downgrade on Strict jobs' wall clock time in the All-Strict+AutoDown configuration. Both the wall-clock time and variation in wall-clock time increase significantly compared to the Strict jobs in the All-Strict configuration. However, as long as the wall-clock time variation can be tolerated by the job, this is a good trade-off because all the jobs in the All-Strict+AutoDown still meet their deadlines and overall job throughput is significantly improved compared to the All-Strict case. Finally, the EqualPart configuration suffers from a high average and variation of wall-clock time, which is caused by the lack of admission control and resource reservation, as well as timesharing.

3.6.2 Impact of Automatic Mode Downgrade

To better understand how automatic mode downgrade improves throughput, we show the detailed execution of each job in the All-Strict and All-Strict+AutoDown for the single-benchmark workload of *bzip2* in Figure 3.7. The x-axes shows time in the number of millions of cycles, while the y-axes shows the first ten accepted jobs. Boxes with solid lines represent the time from a job starting its execution until the time it completes, while boxes with dashed lines represent the amount of time between the job's completion and its deadline. In addition, automatically downgraded jobs are shown in a darker color and the arrows point to the time when they are to be



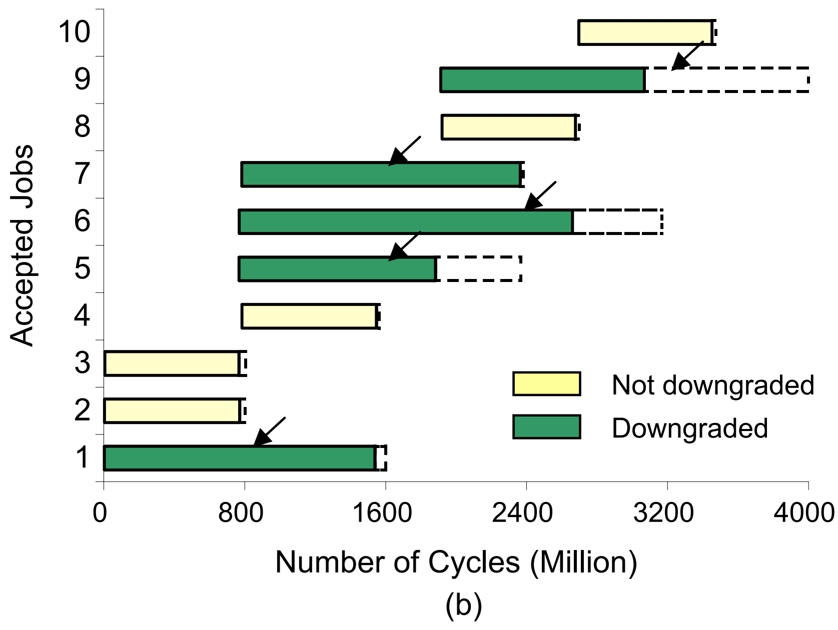


Figure 3.7: Execution trace of ten accepted jobs in the All-Strict case (a) versus in the All-Strict+AutoDown case (b). The jobs are instances of *bzip2*.

switched back to the Strict execution mode.

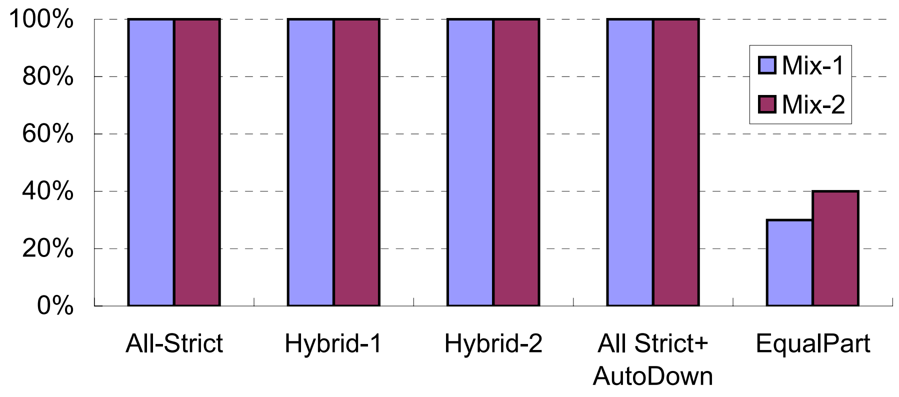
From the figure, we can see that in the All-Strict case, only two jobs can be accepted and run simultaneously, leading to a low admission rate and throughput. It takes 3,883M cycles to complete all ten jobs. In the All-Strict+AutoDown case, it only takes 3,451M cycles to complete ten jobs for several reasons. The first reason is that because Strict jobs running in the Opportunistic mode do not reserve resources, more jobs can be accepted and started earlier. For example, the third job is executed earlier because the first job runs in the Opportunistic mode rather than the Strict mode. The fifth, sixth, and seventh jobs are also admitted sooner and executed earlier. While automatically downgraded jobs run significantly slower, they utilize fragmented resources which otherwise would not have been utilized. The second reason is that when automatically downgraded jobs complete execution, the LAC reclaims their resources, allowing other jobs to be accepted earlier. For example, the completion of the fifth job allows the eighth job to be accepted and executed earlier, while the completion of the sixth job allows the tenth job to be accepted and executed earlier. Another observation is that out of the five automatically downgraded jobs, four of them (the first, fifth, sixth, and seventh) need to be switched back to the Strict mode because they do not complete early enough, while only one (the ninth job) is able to complete before it needs to be switched. Furthermore, the first and seventh jobs likely would not have met their deadlines had resources and timeslot not been reserved. This emphasizes the importance of reserving resources and timeslots for automatically downgraded jobs in order to guarantee meeting their deadlines.

3.6.3 Evaluating Mixed-Benchmark Workloads

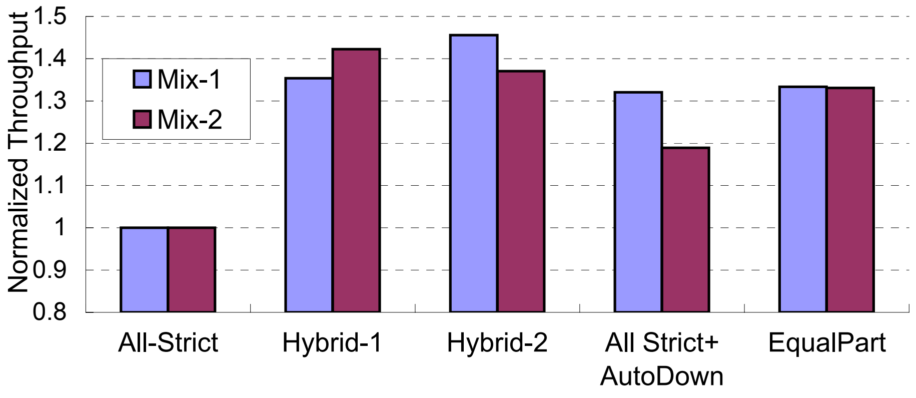
So far we have only used single-benchmark workloads to evaluate our QoS framework. In this section, we evaluate two mixed-benchmark workloads described in Table 3.3. Recall that Mix-1 represents an ideal workload for resource stealing: the cache-sensitive benchmark (*bzip2*) forms Opportunistic jobs while the cache-insensitive benchmark (*gobmk*) forms Elastic(5%) jobs. Mix-2, on the other hand swaps the execution modes of *bzip2* and *gobmk*, so it is not an ideal workload for resource stealing.

Figure 3.8(a) shows the deadline hit rates for different configurations. While our QoS framework achieves 100% deadline hit rates for all Strict and Elastic(X) jobs in mixed-benchmark workloads, EqualPart has low deadline hit rates (30% for Mix-1 and 40% for Mix-2). Figure 3.8(b) shows the job throughput for Mix-1 and Mix-2 normalized to the respective All-Strict cases. Overall, all of Hybrid-1, Hybrid-2, and All-Strict+AutoDown achieve a significant improvement in throughput compared to All-Strict. The throughput improvements achieved in Hybrid-1 and Hybrid-2 sometimes even exceed that of EqualPart. This is a significant result considering that while a majority of jobs miss their deadlines in EqualPart, they meet their deadlines in our QoS framework while simultaneously achieving higher throughputs.

Comparing the throughput improvement between the two workloads, Mix-1 achieves



(a) Deadline hit rates for Strict and Elastic(X) job



(b) Job throughput

Figure 3.8: Comparing two mixed-benchmark workloads.

lower throughput improvement than Mix-2 in Hybrid-1 (35% vs. 42%) but higher than Mix-2 in Hybrid-2 (45% vs. 37%). This is because in Hybrid-1, the Opportunistic jobs can only utilize unallocated cache capacity, and because *bzip2* is more cache sensitive than *gobmk*, instances of *bzip2* have more restricted throughput compared to instances of *gobmk*. However, in Hybrid-2, Opportunistic jobs benefit from extra cache capacity stolen from Elastic(X) jobs. In this case, Opportunistic *bzip2* jobs benefit more from resource stealing than Opportunistic *gobmk* jobs, while at the same time Elastic(X) *gobmk* jobs can give up more of their cache space and are slowed down less than Elastic(X) *bzip2* jobs. The outcome of this combination is that resource stealing is more effective for Mix-1 than for Mix-2. The consequence of this observation is that resource stealing should be applied selectively if our goal is to maximize throughput.

3.6.4 Resource Stealing Techniques Evaluation

The amount of slack available in Elastic(X) jobs determines the amount of cache capacity that can be stolen and reallocated to Opportunistic jobs. In our framework, the slack values of the Elastic(X) jobs are assumed to be specified by the users who may have to consider the trade-off between the amount of stolen cache capacity and the resulting slowdown of the Elastic(X) jobs. One interesting question is whether we could find an appropriate slack value which is not only sufficient to steal unused cache capacity but also remains low in order to minimize the possible slowdown of

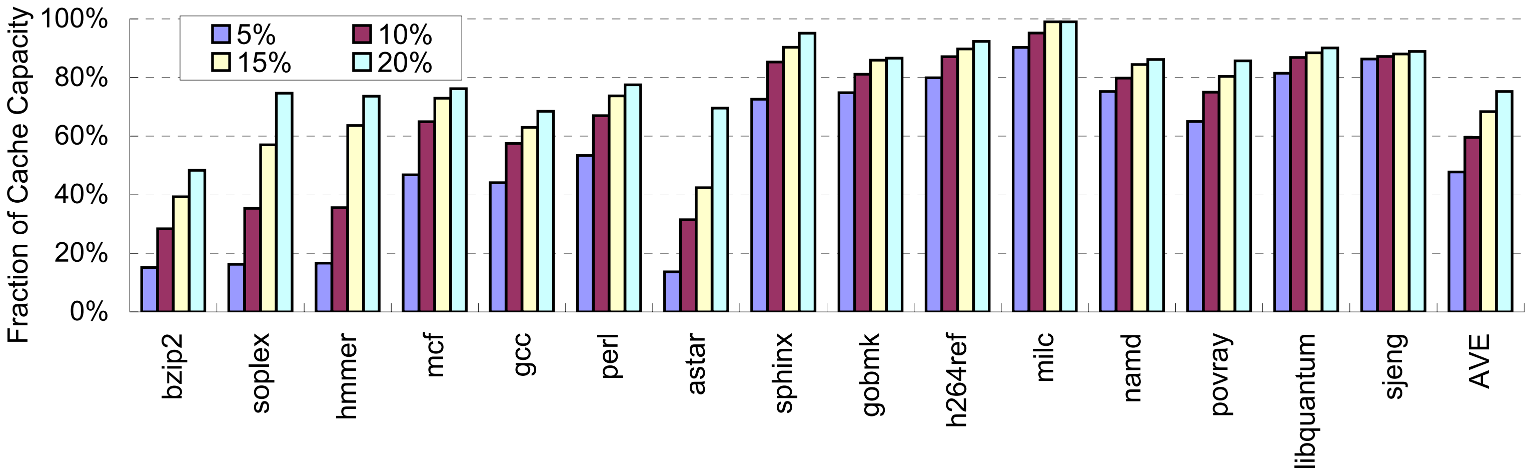


Figure 3.9: The estimated fraction of cache capacity that can be stolen when the performance slack value varies from 5% to 20%. The full evaluation setup can be found in Section 3.5.

the Elastic(X) job. To answer this question, we first calculate the target miss rate increase of each benchmark by using Equation 3.3 for the slack values ranging from 5% to 20%. Based on each benchmark's L2 cache stack distance profile [32], we then estimate the amount of cache capacity that can be reclaimed compared to the initial cache capacity of 896KB (Section 3.5) in order to reach the target miss rate increase for different slack values. The results are presented in Figure 3.9.

From figure 3.9, we can see that different slack values impact the amount of stolen cache capacity non-uniformly across benchmarks. When the slack value is 5%, for *bzip2, soplex, hmmer* and *astar*, the amount of stolen cache capacity is less than 20%, while in other benchmarks, at least 40% cache capacity can be stolen. Note that these four benchmarks actually belong to Group 1 or Gourp 2 (Figure 3.4 in Section 3.5), indicating that they are cache space sensitive benchmarks. Therefore, a smaller cache capacity reduction will cause a 5% CPI increase compared to other cache space insensitive benchmarks. When the slack value increases to 20%, for all benchmarks except *bzip2*, at least 65% of cache capacity can be stolen. An interesting observation is that most benchmarks will not yield a significant amount of extra stolen cache capacity as the slack value increases from 5% to 20% except several cache space sensitive benchmarks in which the amount of stolen cache capacity can be largely increased (e.g. *bzip2* (from 15% to 48%), *soplex* (from 16% to 75%), *hmmer* (from 17% to 74%) and *astar* (from 14% to 70%)). On average, the stolen cache capacity will only increase from 48% to 75% when the slack value increases from 5% to 20%.

Hence, in most cases, a small X value can be sufficient to recover most unused cache capacity and a larger slack value cannot produce considerably more stolen cache capacity which however may cause larger unexpected slowdown for an Elastic(X) job. Overall, the implication of this observation is that the X value of an Elastic(X) job could be a fixed small value other than a specifiable parameter, which simplifies specifying QoS targets. Unless otherwise specified, we will use a slack value of 5% to evaluate different resource stealing algorithms in this section.

Figure 3.10 shows the CPI increase of the Elastic(5%) job for all benchmarks with different resource stealing algorithms. From this figure, we can make several observations. First, the RS_{base} algorithm is not effective in stealing resources for all benchmarks. The actual CPI increase is only between 36% and 76% of the performance slack in *bzip, soplex, mcf, astar* and *sphinx*, and is even lower (less than 20%) in all other benchmarks. Secondly, in quite a few benchmarks (e.g. *bzip2, hmmer, gcc,* etc.), both the RS_{sample} and RS_{safe} algorithms are much more effective in that the CPI increases much closely approach the slack of 5% compared to that of using the RS_{base} algorithm. For most cache space insensitive benchmarks (e.g. *h264ref, milc, namd, libquantum, sjeng,* etc.), the CPI increases are very low for all algorithms and the variation between difference resource stealing algorithms is trivial. This is because the miss rate of those benchmarks cannot be increased as expected no matter how many cache ways are stolen (each job is allocated a minimum of one cache way to avoid starvation). Thirdly, the RS_{sample} algorithm generally outperforms the

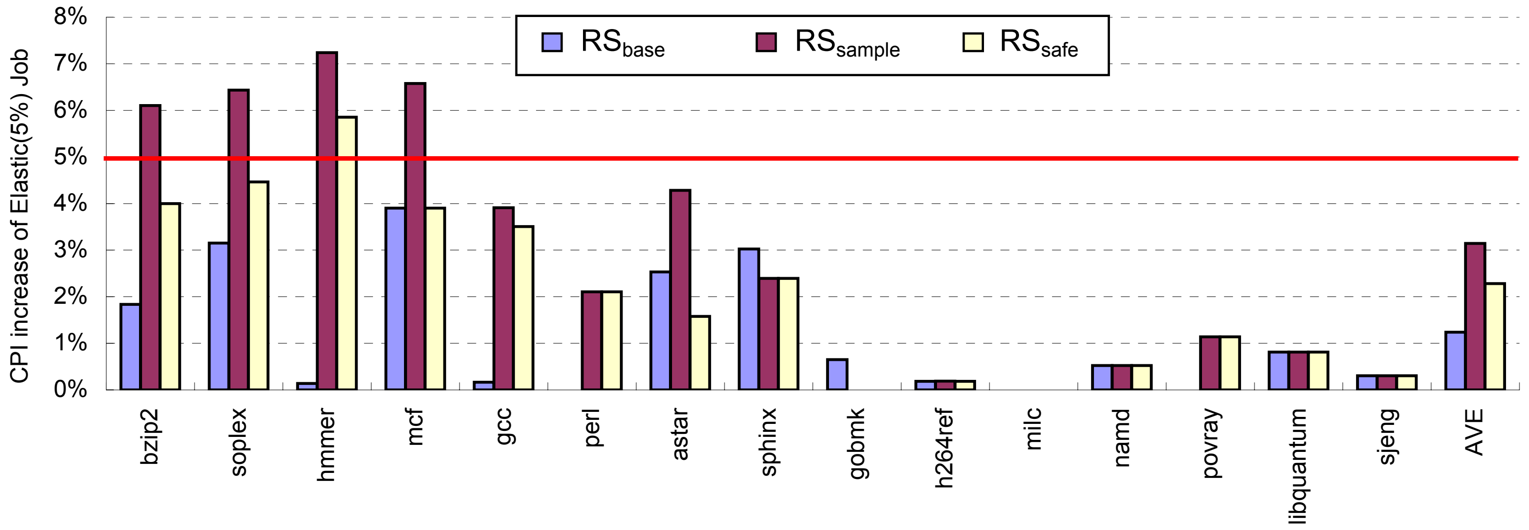


Figure 3.10: The CPI increase of each benchmark when the amount of slack X is 5% with different resource stealing algorithms.

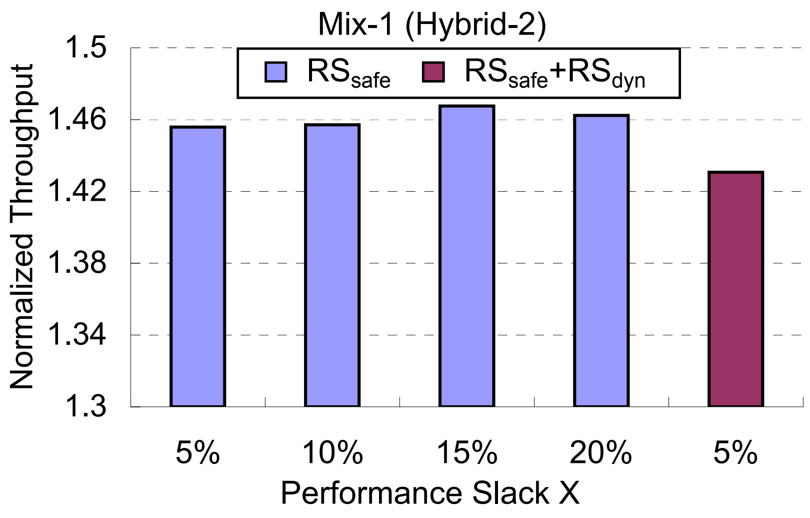
RS_{safe} algorithm in terms of increasing the actual CPI because the latter adds a margin of safety using CPI_{min} which is usually lower than the $\overline{CPI}$ used in the RS_{sample} algorithm. However, the RS_{sample} algorithm often overshoots the maximum allowed slowdown and the actual CPI increases exceed the bound. For example, with RS_{sample} algorithm, the actual CPI increases are 6.1%, 7.2%, 6.4%, 6.7% for *bzip2*, *soplex*, *hmmer* and *mcf* respectively. This is not desirable even with soft QoS requirements. The main reason is that the $\overline{CPI}$ often overestimates the original CPI in the RS_{sample} algorithm for these benchmarks.

That leaves the RS_{safe} algorithm as the best performing algorithm for most benchmarks. It is more effective than RS_{base} in stealing cache capacity to exploit the performance slack, and it does not overshoot the slowdown. In all but one case the CPI increase is less than the specified slack value. The exception is *hmmer*, RS_{safe} overshoots the slowdown by less than 1%. In this case, it turns out that this is because the CPI values collected during sampling intervals (when no resource stealing is applied) are higher than the original CPI. While in both cases no resource stealing is applied, the cache state in sampling intervals has been perturbed by partition changes in prior intervals when resource stealing was applied. This perturbation introduces a slight upward bias in the average sampled CPI values compared to the actual CPI. However, this effect is very small and can be largely mitigated by increasing the size of the repartitioning interval. Another interesting observation is that in *astar, sphinx* and *gobmk*, RS_{base} slightly outperforms RS_{safe} and/or even RS_{sample}. The reason is that in RS_{sample} and RS_{safe}, only two thirds of the overall intervals are spent on resource stealing while RS_{base} attempts to steal resource throughout the whole execution. Although the miss rate increase threshold of RS_{sample} or RS_{safe} is higher, the actual miss rate increase is not significantly higher than that in RS_{base} during the resource stealing intervals. Again, this effect can be largely mitigated by increasing the repartitioning interval size.

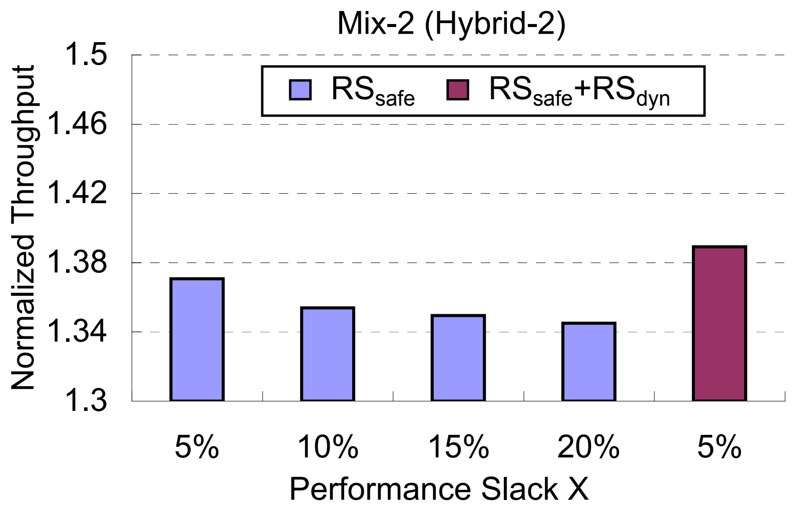
Overall, the proposed RS_{safe} algorithm is much more effective in performing resource stealing compared to RS_{base} while still preserving QoS of Elastic(X) jobs.

3.6.5 Impact of Slack X and RS_{dyn} on Overall Throughput

In previous section, we focus on evaluating the effectiveness of different resource stealing algorithms on stealing cache capacity from Elastic(X) jobs. In this section, we will present how different slack values and the dynamic resource stealing mechanism impact the overall throughput. Figure 3.11 shows the overall throughput of Mix-1 and Mix-2 workload in Hybrid-2 configuration when RS_{safe} is used with different slack values. The result of combining RS_{safe} and RS_{dyn} mechanism when X is 5% is also shown in this figure. The throughput values are normalized to the respective All-Strict cases. From figure 3.11(a), we can see that the throughput improvement can be slightly increased from 45% to 47% when X increases from 5% to 15% because in Mix-1, the opportunistic *bzip*2 jobs keep benefiting from the reallocated cache capacity as the slack value increases, which helps improving the overall throughput



(a)



(b)

Figure 3.11: Normalized throughput of Mix-1 workload (a) and Mix-2 workload (b) in Hybrid-2 configuration when RS_{safe} is used with different slack values and when RS_{safe} is combined with RS_{dyn} mechanism.

to a little extent. However, as X increases to 20%, the throughput improvement is slightly lower than that when X is 15% because the extra speedup of *bzip*2 jobs gained from the extra cache capacity cannot compensate the extra slowdown of Elastic(X) jobs and the extra admission rate reduction due to the longer resource reservation for Elastic(X)jobs. In Mix-2, since the opportunistic *gobmk* jobs, which are cache space insensitive, cannot benefit from the extra cache capacity, the overall throughput improvement is even decreased as the slack value increases (figure 3.11(b)) because resource stealing only makes the Elastic(X) jobs run slower . The implication of these observations is that a small X value is sufficient for improving the overall throughput.

RS_{dyn} mechanism impacts the overall throughput of Mix-1 and Mix-2 differently. For Mix-1, when RS_{dyn} is applied, the throughput improvement is actually lower than that of not applying RS_{dyn} (43% vs. 45%). This is because RS_{dyn} suspends the resource stealing in the first several repartitioning intervals until the reduction of total number of cache misses is larger than 5%. During this period, *bzip*2 jobs cannot steal extra cache capacity. This performance gap however can be largely reduced if Elastic(X) jobs' wall-clock time is increased. For Mix-2, resource stealing is useless since the opportunistic *gobmk* jobs are cache space insensitive. As we expect, using RS_{dyn} achieves higher throughput improvement (39% vs. 37%) because the resource stealing is suspended during most of execution time. In general, RS_{dyn} mechanism can be used to achieve higher overall throughput without user's judgement of whether resource stealing is useful or not.

3.6.6 Characterization of the Local Admission Controller

In our framework, the LAC is implemented as a user level program and is fully simulated in our evaluation. The LAC incurs performance overheads when it performs admission tests and scheduling. However, since the LAC only performs a simple admission control policy and implements a simple scheduling algorithm, the occupancy of the LAC is less than 1% of each workload's wall-clock time. If the number of jobs submitted to the CMP increases, or if the number of cores in the CMP increases, the LAC's overheads will increase proportionally although they likely remain low.

3.7 Related Work

In a CMP system, some platform resources, such as the off-chip bandwidth and the lowest level on-chip cache, are typically shared among cores. With the increasing number of cores on a chip (possibly to more than one hundred cores by 2015 [3]), the contention for these critical shared resources suffered by applications running simultaneously in different cores will increase significantly and needs to be carefully managed.

Some studies that address the management of shared resources have focused on improving the overall throughput or fairness of the CMP. Suh et al. proposed a cache partitioning policy that minimizes the total number of cache misses [49], Kim et al. proposed a cache partitioning policy that optimizes for uniform slowdown (fairness) to applications that share the cache [25], while Qureshi et al. proposed utility-based cache partitioning [36]. Hsu et al. studied the impact of various optimization goals in guiding how cache partitions are allocated in a CMP architecture [20]. The optimization goals include maximizing the overall performance metric (e.g. IPC or miss rate) and the overall fairness. Cho and Jin proposed an OS-level page allocation algorithm in a shared L2 non-uniform cache architecture for future many-core processors to reduce the cache access latency, on-chip network traffic and power consumption [10]. Chang and Sohi proposed a cooperative cache partitioning algorithm that optimizes for several metrics, such as the sum of per-thread slowdowns as well as the harmonic mean of per-thread speedups over an equal-partition cache baseline [8]. While these studies seek to mitigate the impact of contention and optimize for an overall goal, they do not provide QoS to individual applications.

Some recent studies have recognized the need for CMPs to have QoS-enabling features to provide differentiated services to various applications. For example, Iyer described the need for a priority-based QoS framework in CMP architectures in which a job can specify whether it should be run with a high or low priority, and resource allocation is guided by job priorities [21]. Rafique et al. proposed an architecture support and Operating System (OS) interface that allows OS-level cache partitioning, in which an application is prevented from occupying more than a certain fraction (quota) of the cache [37]. Nesbit et al. proposed a Virtual Private Cache (VPC) that combines the resource allocation policies for caches and the memory controller using a fair queuing algorithm [33]. VPC provides an abstraction of private caches through partitioning the shared cache into per-core partitions, while resource allocation policies ensure that the IPC achieved is equal to that of real private caches. Iyer et al. [22] proposed several priority-based resource management policies as well as a QoS-aware cache and memory architecture. Individual applications can specify their own QoS target (e.g. IPC, miss rate, cache space) and the hardware dynamically adjusts cache partition sizes to meet their QoS targets. Lin et al. [29] proposed a software based experimental methodology to implement and evaluate several cache partitioning policies that optimize for fairness and QoS. In this study, the QoS target of an application is specified in IPC and an OS based cache partitioning scheme is used to emulate the dynamic partitioning of hardware cache in order to reach the application's target IPC. We note that while all above studies mention QoS, they do not associate QoS with the notion of performance guarantee. While both priorities and cache quota correlate with performance and help the system to favor one application over another in allocating resources, they do not automatically provide QoS guarantees to the applications.

In some QoS models, individual applications can specify an acceptable performance level expressed in IPC. The system then translates the IPC into the minimum resources required to achieve it, through profiling a thread at run-time and recording how the thread's IPC changes as the amount of resources are varied [27, 56]. The fact that the studies above require a greedy search in the resource allocation space illustrates the difficulty of using IPC as a QoS target. We believe that to really provide QoS, the CMP must have an ability to easily compare available and demanded computation capacity, and such an ability is a critical component that enables the construction of an admission control policy.

Finally, some of the concepts in our framework, such as the resource reservation and admission control, are borrowed from the real-time system domain [4, 5, 39]. However, while in traditional real-time systems the operating system and processor architecture are structured to suit the needs of real-time constraints, we seek to provide QoS in CMP-based general purpose servers with a largely unmodified OS, processor architecture, and memory hierarchy.

3.8 Conclusions

This chapter has presented a QoS framework that provides performance QoS through the use of appropriate QoS target specification and execution modes. It has also presented how throughput can be improved while still preserving QoS, by employing several techniques such as manual and automatic mode downgrade and resource stealing. Through this study, we discover several findings. First, QoS targets should be specified with Resource Usage Metrics (RUM) in order to fully provide QoS and to build an admission control policy. Secondly, QoS-enabling features such as the ability to dynamically partition caches and a resource manager which tries to meet the IPC target of all jobs, are by themselves insufficient for fully providing QoS. Thirdly, substantial throughput is lost when we provide strict QoS with our framework due to external processor core and cache fragmentation, and internal cache fragmentation. Fourthly, the two alternative QoS execution modes (Elastic(X) and Opportunistic) enable the system to recover much of the lost throughput by reducing resource fragmentation. Manual mode downgrade in general is more effective than automatic mode downgrade if there is an appropriate mixture of Elastic(X) and Opportunistic jobs. However, even when there are only Strict jobs, the overall throughput of the system can still be boosted by using automatic mode downgrade. Finally, resource stealing is an effective microarchitecture technique for improving throughput by reallocating excess cache capacity from Elastic(X) jobs to Opportunistic jobs while still meeting the Elastic(X) jobs' QoS targets. Among all the proposed resource stealing algorithms,

RS_{safe} is most effective in performing resource stealing while still preserving QoS. We also find that a small X value is sufficient to recover most unused resource and improve throughput. In addition, dynamic resource stealing mechanism can be used to achieve higher overall throughput without user's understanding of the characteristics of the workload.

Chapter 4

Conclusions

The scaling, power, thermal and reliability constraints of uniprocessors have turned the focus of the computer industry towards chip muti-processor (CMP) systems in many computing domains. Within a decade, we can expect that tens of processor cores will be able to be integrated into a single chip which supports hundreds of concurrent threads. However, since some important platform resources such as on-chip cache and off-chip bandwidth are shared by all processor cores, one fundamental factor that determines the scalability of the CMP architecture is whether the CMP can efficiently manage the shared resources to ensure that all concurrent threads can make good progress as the total number of concurrent threads increases. This challenge is also known as the resource sharing problem which has become a significant issue in recent CMP architecture research. In this dissertation, we have focused on one important shared platform resource, the lowest level on-chip cache in a CMP architecture, and have addressed the cache sharing problem from two aspects. The major contributions and conclusions are summarized as follows.

Firstly, we have presented three tractable models (FOA, SDC and *Prob*) to predict the impact of cache sharing. Such impact was considered to be intractable because the factors that influence the cache sharing impact were not known. Two models (FOA and SDC) are based on heuristics. They are simple to use but hardly accurate. On the other hand, the analytical inductive probability model (*Prob*) which realizes a simple Markov process is very accurate, achieving an average absolute error of 3.5%. In addition to not only offering accurate prediction, the *Prob* model also provides a practical tool through which we can investigate the factors that influence the impact of cache sharing and reveal certain non-obvious interactions between different co-scheduled threads. One particular insight that we have gained is that for a base thread, the number of cache misses and the stack distance profile shape of the interfering thread largely determine the base thread's cache miss increase. This knowledge could be very useful for future thread scheduler design in identifying and avoiding suboptimal thread co-schedules.

Secondly, we have presented a Quality of Service (QoS) framework which provides performance guarantee for individual applications. The ability to provide performance guarantee is a very favorable feature in CMP-based servers, especially for the applications in utility computing and virtualization domains. However, providing such ability in servers is really challenging. The major obstacles are how to enable a CMP to check whether a job's QoS target can be satisfied or not and how to control the job admission when the incoming jobs are irregular. In this dissertation, we have argued that the job's QoS target should be specified through resource capacity requirements in order to easily build an admission control policy. The admission controller ensures that jobs are accepted only when their QoS targets can be met. However, resource capacity specification may lead to jobs overspecifying their QoS targets. Such QoS overspecification can easily lead to resource fragmentation and result in throughput reduction. We have presented two techniques to recover the lost throughput. The first one relies on the speculative downgrade of a job's QoS execution modes (from Strict mode to Elastic(X) mode or from Strict mode to opportunistic mode). The second technique is referred to as resource stealing technique, which discovers and re-allocates the excess resources from the specified jobs while still preserving their QoS targets. We have investigated three resource stealing algorithms (RS_{base}, RS_{sample} and RS_{safe}) and have compared them in terms of safety and effectiveness. Through simulation, we found that by using the proposed QoS framework and throughput improvement techniques, a CMP can ensure that all accepted jobs have their QoS targets satisfied and the overall throughput can be improved between 13% and 45%, which is significantly closer to a non-QoS CMP. In addition, we found that a small performance slack value is sufficient to achieve effective resource stealing, so a fixed small X value can be used for Elastic(X) jobs, which simplifies specifying the QoS targets.

Overall, in this dissertation, we have investigated methodology, techniques, and algorithms to analyze and manage the shared cache in a CMP architecture. We can conclude that the traditional heuristics assuming that factors like IPC, miss rates or instruction types impact the cache contention are misleading. The cache contention is actually largely affected by the temporal reuse behaviors of the co-scheduled applications. We can also conclude that combining QoS target specification, QoS execution modes, job admission control, job scheduling, and resource stealing is essential towards providing QoS in a CMP architecture. Finally, the proposed QoS framework could additionally serve as a *base framework* for future research work as it associates the QoS with the notion of performance guarantee.

Bibliography

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

[2] Jean Loup Baer and Wen Han Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proc. of the International Symposium on Computer Architecture*, pages 73–80, 1988.

[3] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski, and Justin Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. *Technology@Intel Magazine*, 2005.

[4] Bjorn B. Brandenburg and James H. Anderson. Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors. In *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, 2007.

[5] Giorgio Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[6] 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.

[7] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multiprocessor Architecture. In *Proc. of the 11th International Symposium on High Performance Computer Architecture*, pages 340–351, 2005.

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

[9] S. Chatterjee, E. Parker, P.J. Hanlon, and A.R. Lebeck. Exact Analysis of the Cache Behavior of Nested Loops. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, 2001.

[10] Sangyeun Cho and Lei 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.

[11] H. Dybdahl and P. Stenstrm. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proc. of the 13th International Symposium on High Performance Computer Architecture*, pages 2–12, 2007.

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

[13] B.B. Fraguela, R. Doallo, and E.L. Zapata. Automatic Analytical Modeling for the Estimation of Cache Misses. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, 1999.

[14] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703-746, 1999.

[15] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. *ACM SIGARCH Computer Architecture News*, 35(1):21–30, 2007.

[16] Fei Guo and Yan Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 228–239, 2006.

[17] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 343–356, 2007.

[18] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 3rd edition, 2003.

[19] S. Hily and A. Seznec. Contention on the 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. *IRISA Technical Report 1086*, 1997.

[20] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari 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*, pages 13–22, 2006.

[21] Ravishankar 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.

[22] Ravishankar Iyer, Li Zhao, Fei Guo, Yan Solihin, Srihari Markineni, Don Newell, Rameshi Illikkal, Lisa Hsu, and Steven 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.

[23] J. Renau, et al. SESC. *http://sesc.sourceforge.net*, 2004.

[24] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proc. of the 10th International Symposium on High Performance Computer Architecture* , pages 288–299, 2004.

[25] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning on a Chip Multi-Processor Architecture. In *Proc. of the International Conference on Parallel Architecture and Compilation Techniques*, pages 111–122, 2004.

[26] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code on an Intelligent Memory Architecture. In *Proc. of the 7th International Symposium on High Performance Computer Architecture*, pages 121–132, 2001.

[27] Jae W. Lee and Krste Asanovic. METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 135–147, 2006.

[28] T. Leng, R. Ali, and J. Hsieh. A Study of Hyper-Threading in High-Performance Computing Clusters. *Dell Power Solutions HPC Cluster Environment*, pages 33–36, 2002.

[29] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P.Sadayappan. Gaining Insights into Multicore Cache Partitioning: Briding the Gap between Simulation and Real Systems. In *Proc. of the 14th International Symposium on High Performance Computer Architecture.*, pages 367–378, 2008.

[30] Yong Luo, Olaf M. Lubeck, Harvey Wasserman, Federico Bassetti, and Kirk 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.

[31] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt

Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

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

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

[34] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing: Introduction. *Communications of the ACM*, 46(10):24–28, 2003.

[35] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 167–178, 2006.

[36] Moinuddin K. Qureshi and Yale 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*, pages 423–432, 2006.

[37] Nauman Rafique, WonTaek Lim, and Mithuna Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[38] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[39] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernel: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998.

[40] Parthasarathy Ranganathan and Norman Jouppi. Enterprise IT Trends and Implications for Architecture Research. In *Proc. of the 11th International Symposium on High Performance Computer Architecture*, pages 253–256, 2005.

[41] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. of the International Symposium on Computer Architecture*, pages 336–349, 2003.

[42] A. Snavely and D. Tullsen. Symbiotic Jobs Scheduling for a Simultaneous Multithreading Processor. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[43] Y. Solihin, V. Lam, and J. Torrellas. Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. In *Proc. of International Conference on Supercomputing*, 1999.

[44] Standard Performance Evaluation Corporation. Spec cpu2000. *http://www.spec.org/cpu2000*, 2000.

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

[46] Harold Stone, John Turek, and Joel Wolf. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[47] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–396, 2006.

[48] 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 117–128, 2002.

[49] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of the 15th International Conference on Supercomputing*, pages 1–12, 2001.

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

[51] R. Turner and W. D. Strecker. Use of the LRU Stack Depth Distribution for Simulation of Paging Behavior. *Communications of the ACM*, 20(11):795–798, 1977.

[52] X. Vera and J. Xue. Let's Study Whole-Program Cache Behaviour Analytically. In *Proc. of International Symposium on High Performance Computer Architecture*, pages 175–186, 2002.

[53] D. Vianney. Hyper-Threading Speeds Linux: Multiprocessor Performance on a Single Processor. *IBM DeveloperWorks*, 2003.

[54] J. Wang, S. Zhou, and W. Ahmed. Lsbatch: a Distributed Load Sharing Batch System. *Technical Report CSRI-286, Univ. of Toronto*, 1993.

[55] H. J. Wassermann, O. M. Lubeck, Y. Luo, and F. Bassetti. Performance Evaluation of the SGI Origin2000: A Memory-Centric Characterization of LANL ASCI Applications. In *Proc. of Supercomputing*, pages 1–11, 1997.

[56] Thomas Y. Yeh and Glenn Reinman. Fast and Fair: Data-Stream Quality of Service. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 237–248, 2005.