Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture *

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{dchandr, fguo, skim16, solihin}@ece.ncsu.edu

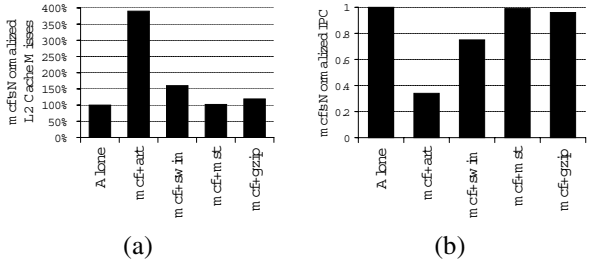
Abstract

This paper studies the impact of L2 cache sharing on threads that simultaneously share the cache, on a Chip Multi-Processor (CMP) architecture. Cache sharing impacts threads non-uniformly, where some threads may be slowed down significantly, while others are not. This may cause severe performance problems such as sub-optimal throughput, cache thrashing, and thread starvation for threads that fail to occupy sufficient cache space to make good progress. Unfortunately, there is no existing model that allows extensive investigation of the impact of cache sharing. To allow such a study, we propose three performance models 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 profile 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 for each thread due to cache sharing. The models differ by their complexity and prediction accuracy. We validate the models against a cycle-accurate simulation that implements a dual-core CMP architecture, on fourteen pairs of mostly SPEC benchmarks. The most accurate model, the Inductive Probability *model, achieves an average error of only 3.9%. Finally, to demonstrate the usefulness and practicality of the model, a case study that details the relationship between an application's temporal reuse behavior and its cache sharing impact is presented.*

1. Introduction

In a typical Chip Multi-Processor (CMP) architecture, the L2 cache and its lower level memory hierarchy are shared by multiple cores [8]. Sharing the L2 cache allows high cache utilization and avoids duplicating cache hardware resources. However, as will be demonstrated in this paper, cache sharing impacts threads non-uniformly, where some threads may be slowed down significantly, while others are not. This may cause severe performance problems such as sub-optimal throughput, cache thrashing, and thread starvation for threads that fail to occupy sufficient cache space to make good progress.

To illustrate the performance problem, Figure 1 shows the number of L2 cache misses (1a) and IPC (1b) for *mcf* when it runs alone compared to when it is co-scheduled with another thread which runs on a different processor core but sharing the L2 cache. The bars are normalized to the case where *mcf* runs alone. The figure shows that when *mcf* runs together with *mst* or *gzip*, *mcf* does not suffer from many additional misses compared to when it runs alone. However, when it runs together with *art* or *swim*, its number of misses increases to roughly 390% and 160%, respectively, resulting in IPC reduction of 65% and 25%, respectively.



(a) (b)

Figure 1. The number of L2 cache misses (a), and IPC (b), for *mcf* when it runs alone compared to when it is co-scheduled with another thread. The L2 cache is 512KB, 8-way associative, and has a 64-byte line size.

Past studies have investigated profiling techniques that *detect* but *not prevent* the problems, and apply inter-thread cache partitioning schemes to improve fairness [11] or throughput [22, 11]. However, the questions of what factors influence the impact of cache sharing suffered by a thread in a co-schedule, and whether the problems can be predicted (hence prevented) were not addressed. This paper addresses both questions by presenting analytical and heuristic models to predict the impact of cache sharing.

Past performance prediction models only predict the number of cache misses in a uniprocessor system [3, 4, 5, 6, 12, 24, 26], or predict cache contention on a single processor time-shared system [1, 21, 23], 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 paper goes beyond past studies and presents three

*This work is supported in part by the National Science Foundation through grant CNS-0406306 and Faculty Early Career Development Award CCF-0347425, and by North Carolina State University.

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. 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 study gives an insight into what types of applications are vulnerable (or not vulnerable) to a large increase in cache misses under sharing.

The rest of the paper is organized as follows. Section 2 presents the three models. Section 3 details the validation setup for our models. Section 4 presents and discusses the model validation results and the case study. Section 5 describes related work. Finally, Section 6 summarizes the findings.

2. 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), then it presents an overview of the three models (Section 2.2), the *frequency of access* (FOA) model (Section 2.3), the *stack distance competition* (SDC) model (Section 2.4), and the *inductive probability* (*Prob*) model (Section 2.5). Since the most accurate model is *Prob*, the discussion will focus mostly on *Prob*.

2.1. 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. [15]. This is beyond the scope of this paper.

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 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 paper are likely to be applicable to other implementations as well.

2.2. Model Overview

2.2.1. 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 [14, 3, 12, 19], and is sometimes also referred to as marginal gain counters [21, 22]. 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 [3], by simulation, or by running the thread alone in the system [22].

Figure 2 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. 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

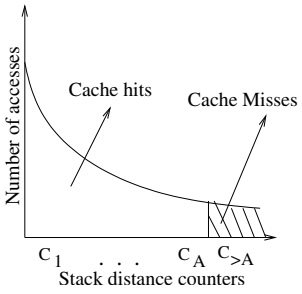


Figure 2. Illustration of a stack distance profile

cache that has A' associativity, where $A' < A$, the new number of misses can be computed as:

$$miss = C_{>A} + \sum_{i=A'+1}^{A} C_i \quad (1)$$

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 call 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).

2.2.2. 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 4.4, 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.

2.2.3. 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 3. Figure 3a illustrates how FOA and SDC predict the number of extra cache misses. Using a set of heuristics, they first predict

Table 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

the effective cache space (A') that a thread will have under cache sharing. Then A' is input into Equation 1 to predict the number of misses that the thread will suffer under cache sharing (the shaded region in Figure 3a). 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 3b.

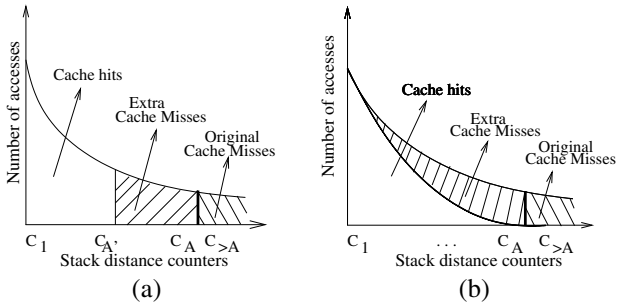


Figure 3. Comparing the prediction approach of FOA and SDC (a), with that of *Prob* (b).

Table 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 2.5. From the table, we expect *Prob* to be the most accurate because it takes into account all the factors.

2.3. 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 share the cache. The effective cache size for thread X can be calculated from:

$$effCacheSize_X = \frac{Af_X}{\sum_{j=1}^{N} Af_j} \times CacheSize \quad (2)$$

By taking $A' = \frac{effCacheSize_X}{numCacheSet}$, and plugging it into Equation 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.

2.4. Stack Distance Competition (SDC) Model

The *stack distance competition* (SDC) model tries to construct a new stack distance profile that merges individual stack distance profiles of threads that run together, by taking a subset of counters from the individual profiles. A merging algorithm iteratively selects a stack distance counter from a "winning" profile to be included in the merged profile, until the merged profile fills up all of its hit counters (i.e. $C_1, C_2, \ldots, C_A$). To achieve this, each individual profile is assigned a current pointer that is initialized to the first stack distance counter. In each iteration, all the counters pointed by the current pointers from all individual stack distance profiles are compared, and the profile with the highest counter value is selected as the winner. The winner's counter is copied into the merged profile, and its current pointer is advanced. After the last iteration, the effective cache space for each thread is computed proportionally to the number of stack distance counters that are included in the merged profile.

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.

2.5. 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.*

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 4. 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 ($cseq(1, 2)$).

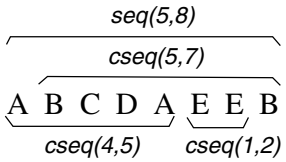


Figure 4. Illustration of the relationship between a sequence and circular sequences.

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.

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*

[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.

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 $d_X \leq A$.*

Let us now consider the impact of running a thread X together with another thread Y that shares the L2 cache with it. Figure 5 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 $d_Y > A - d_X$.

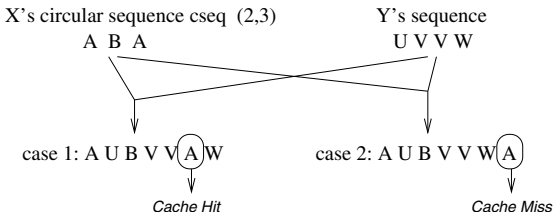


Figure 5. Illustration of how intervening accesses from another thread determines whether the last access of a circular sequence will be a cache hit or a miss. Capital letters in a sequence represent line addresses. The figure assumes a 4-way associative cache and all accesses are to a single cache set.

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

$$P_{miss}(cseq_X(d_X, \overline{n_X})) = \sum_{d_Y = A - d_X + 1}^{E(n_Y)} P(seq_Y(d_Y, E(n_Y))) \quad (3)$$

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 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.

[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.

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

We will now describe how each step is performed.

2.5.1. Step 1: Computing $\overline{n_X}$

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

$$\overline{n_X} = \frac{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X)) \times n_X}{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X))} \quad (4)$$

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.

2.5.2. 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 with the access frequency per set of thread X (Af_X):

$$T(cseq_X(d_X, \overline{n_X})) = \frac{\overline{n_X}}{Af_X} \quad (5)$$

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:

$$E(n_Y) = Af_Y \times T(cseq_X(d_X, \overline{n_X})) \quad (6)$$

2.5.3. 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 $P(seq(d, n))$.

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 $seq(d, n)$ represents 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 $\sum_{i=1}^{d} P(cseq(i, *))$, 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 to occur, $P(seq(d-1, n-1)) = 0$. 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.

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

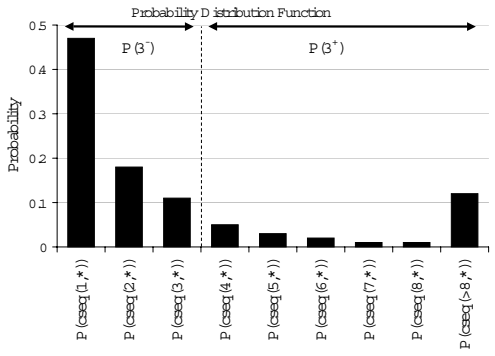


Figure 6. Example probability distribution function that shows $P(3^-)$ and $P(3^+)$. The function is computed by using the formula in Corollary 2.

Corollary 2 and Figure 6 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 $P(3^+)$.

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

2.5.4. 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 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 ($C_{>A}$):

$$miss_X = C_{>A} + \sum_{d_X=1}^{A} P_{miss}(cseq_X(d_X, \overline{n_X})) \times C_{d_X} \quad (7)$$

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 [9]. 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 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 [10]. Unless noted otherwise, the L2 cache only stores data and does not store instructions.

Table 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 [20]; and *mst* from Olden benchmark suite. Table 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.

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.

Table 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%
gzip	test	3%
mcf	test	9%
perlbmk	reduced ref	59%
swim	test	75%
mst	1024 nodes	63%

4. Evaluation and Validation

This section will discuss four sets of results: the impact of cache sharing on IPC (Section 4.1), validation of the prediction models (Section 4.2), sensitivity study (Section 4.3) and a case study on the relationship between temporal reuse behavior and the impact of cache sharing (Section 4.4).

4.1. Impact of Cache Sharing

Figure 7 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 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*,

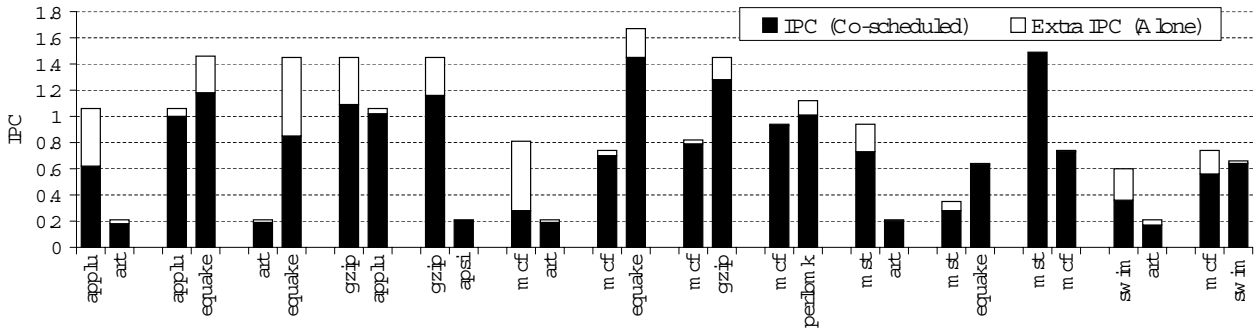


Figure 7. The impact of cache sharing on the IPCs of co-scheduled threads.

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.

4.2. Model Validation

Table 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 4.2.1. 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 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 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.

4.2.1. Remaining Inaccuracy

To further validate the *Prob* model, we relax two assumptions that we have made in Section 2.1, namely the multi-level cache inclusion, and the unified L2 cache. The last two columns in Table 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

Table 4. Validation Results. In each co-schedule, the benchmark that finishes to completion is indicated with an asterisk.

Co-schedule		Number of instructions executed	Extra L2 Cache Misses Due to Sharing	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%
Avg: Arithmetic Mean of Absolute Error: $\frac{\sum \|E_j\|}{n}$				**18.6%**	**13.2%**	**3.9%**	4.3%	4.7%
Geom: Geometric Mean of Absolute Error: $(\Pi(1+\|E_j\|))^{\frac{1}{n}} - 1$				**13.3%**	**11.6%**	**3.7%**	4.1%	4.4%

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 frequencies, 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 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 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.

4.3. 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 5 and Table 6, respectively.

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 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 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%

Table 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.

Table 6. Prediction accuracy for different cache associativities.

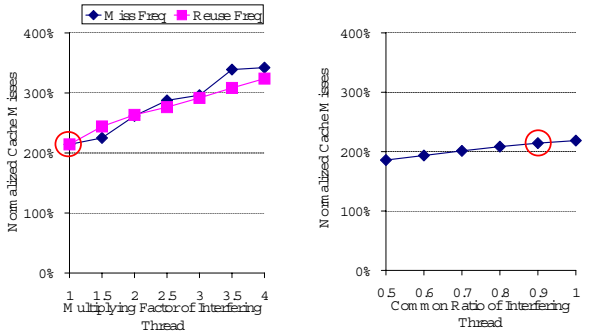
Cache Size, Assoc		Extra L2 Cache misses	L2 Miss Prediction Absolute Error (E_j)		
			FOA	SDC	*Prob*
512KB, 4	min	0%	0%	0%	0%
	max	361%	80%	78%	45%
	avg	47%	12.2 %	13.4%	7.2%
512KB, 8	min	0%	0%	0%	0%
	max	296%	264%	74%	25%
	avg	39%	18.6 %	13.2%	3.9%
512KB, 16	min	0%	0%	0%	0%
	max	275%	69%	73%	36%
	avg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

4.4. Case Study: Relationship Between Temporal Reuse Behavior and Cache Sharing Impact

In this case study, we evaluate how temporal reuse behavior affects the impact of cache sharing by generating synthetic stack distance profiles that densely cover a large range of temporal reuse behavior. To do that, we choose a *base thread* and vary the temporal reuse behavior of an *interfering thread*.

For the base thread, its stack distance profile is synthesized using a geometric progression: $C_1 = Z, C_2 = Zr, C_3 = Zr^2, \ldots, C_i = Zr^{i-1}$, where Z denotes the *amplitude*, and $0 < r < 1$ denotes the *common ratio* of the progression. This is in general a reasonable approximation to an application's stack distance profile because more recently used lines are more likely to be reused than less recently used lines. We also choose $C_{>A} = \sum_{i=A}^{\infty} Zr^i = \frac{Zr^A}{1-r}$. We perform three experiments, where in each experiment, we vary only one of the three factors of the interfering thread that affects the impact of cache sharing (Section 2.2.2).

In the first experiment, we only vary the reuse frequency of the interfering thread, by substituting the amplitude of the interfering thread's geometric progression with a new one, $Z' = kZ$, where the multiplying factor $k = 1, 1.5, 2, \ldots, 4$. In the second experiment, we only vary the miss frequency of the interfering thread, by substituting the interfering thread's miss frequency with a new one, $C'_{>A} = k \times C_{>A}$, where the multiplying factor $k = 1, 1.5, 2, \ldots, 4$. In the third experiment, we vary the shape of the stack distance profile, by substituting the interfering thread's common ratio with a new one, $r' = 0.5, 0.6, 0.7, 0.8, 0.9, 1$, while keeping the reuse and miss frequencies constant.



(a) (b)

Figure 8. The impact of cache sharing to a base thread with a flat stack distance profile ($r = 0.9$).The circles indicate the case where the interfering and the base threads are identical.

Figure 8 shows the result of the three experiments when the base thread's common ratio is 0.9, indicating a mostly flat stack distance profile. We vary the interfering thread's miss and reuse frequency in the x-axes of Figure 8a, and the interfering thread's common ratio in the x-axes of Figure 8b. The y-axes of the figure shows the number of extra cache misses under cache sharing normalized to the amplitude

(Z). The circles indicate the case where the parameters of the interfering and the base threads are identical.

The figure shows that the number of extra misses of the base thread increases almost linearly with the interfering thread's higher miss and reuse frequencies. This is because the base thread's effective cache space is reduced, and due to the flat stack distance profile, the number of misses of the base thread increases sharply. In terms of the interfering thread's common ratio, the more concentrated the interfering thread's stack distance profile (smaller common ratio), the fewer extra cache misses the base thread suffers. This is because if the interfering thread has a more concentrated stack distance profile, it retains fewer cache lines that it brings in the cache, giving more effective cache space to the base thread. However, the impact of the common ratio of the interfering thread is a lot less significant than the impact of the miss and reuse frequencies of the interfering thread.

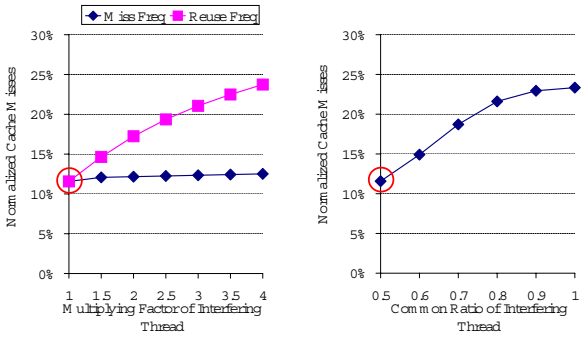


Figure 9. The impact of cache sharing to a base thread with a concentrated stack distance profile ($r = 0.5$). The circles indicate the case where the interfering and the base threads are identical.

Figure 9 is similar to Figure 8 except that the base thread's common ratio is 0.5, indicating a very concentrated stack distance profile. This means the base's thread working set is smaller but reused very frequently. The figure shows that the base thread is not affected at all by the miss frequency of the interfering thread. This is because the base thread reuses its working set frequently, causing most of the interfering thread's cache misses to replace its own lines. However, if the interfering thread's reuse frequency increases, the base thread is unable to keep its working set in the cache, and suffers from a large cache miss increase. The same holds true when the interfering thread's common ratio is increased. Apparently, with a flatter stack distance profile, the interfering thread increases its effective cache space, incurring a sharp increase in the base thread's number of extra cache misses. However, up to a certain point, the interfering thread cannot increase its effective cache space much more, and the base thread's number of extra cache misses stabilizes.

Comparing Figure 9 and Figure 8, we can make a few observations. First, when the base thread's stack distance profile is more concentrated, it suffers less impact from cache sharing (10-25% extra cache misses vs. 200-400% extra cache misses). In addition, the impact of cache sharing on the base thread is significantly determined by the temporal reuse behavior of the interfering thread. The interaction between the base and interfering threads is sometimes not easily obvious, illustrating the need of the *Prob* model in understanding them. Finally, the figures explain why in Table 4 some applications are (or are not) vulnerable to a large increase in the number of cache misses under cache sharing. For example, *gzip* and *mcf* are very vulnerable to a large increase in the number of cache misses under cache sharing because their stack distance profiles are much flatter than other applications. This is due to *mcf*'s and *gzip*'s good temporal reuse for a high number of LRU stack positions. Applications that are not vulnerable to this effect are ones with concentrated stack distance profiles. In addition, it also explains the variability of the impact of cache sharing on a single application. For example, since *applu* has a higher reuse and miss frequency than *apsi*, *gzip* suffers a lot more extra cache misses in *gzip+applu* compared to in *gzip+apsi*.

5. Related Work

Previous performance prediction models only predict the number of cache misses in a uniprocessor system [24, 3, 5, 6, 4, 26, 12], or predict cache contention on a single processor time-shared system [21, 23, 1]. 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 paper 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 a SMT or CMP system in past studies [7, 22, 10, 25, 7, 13], 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 [17, 16]. 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.

Suh, et al. [22] and Kim, et. al. [11] 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 scheduler, and their schemes can optimize the performance of a selected co-schedule further. This remains a future work.

Finally, the model proposed by Wasserman et al. predicts the average cache miss penalty of a program on a superscalar uniprocessor [26], while that proposed by Solihin et al. predicts the miss penalty on a multiprocessor system [18]. Integrating such models with *Prob* allow a full performance model that predicts the impact of cache sharing on IPCs of co-scheduled threads.

6. Conclusions

This work has studied 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.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. *ACM Trans. on Computer Systems*, 1989.

[2] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proc. of the Intl. Symp. on Computer Architecture*, 1988.

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

[4] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proc.of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2001.

[5] B. Fraguela, R. Doallo, and E. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proc.of Intl. Conf. on Parallel Architectures and Compilation Techniques*, 1999.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703-746, 1999.

[7] S. Hily and A. Seznec. Contention on the 2nd level cache may limit the effectiveness of simultaneous multithreading. *IRISA Tech. Rep. 1086*, 1997.

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

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

[10] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2004.

[11] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multi-processor architecture. In *Proc. of the Intl. Conf. on Parallel Architecture and Compilation Techniques*, 2004.

[12] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code on an Intelligent Memory Architecture. In *7th Intl. Symp. on High Performance Computer Architecture*, 2001.

[13] 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.

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

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, 2003.

[16] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[17] A. Snavely, et al. Explorations in symbiosis on two multithreaded architectures. In *Workshop on Multithreaded Execution, Architecture, and Compilation*, 1999.

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

[19] Y. Solihin, J. Lee, and J. Torrellas. Automatic Code Mapping on an Intelligent Memory Architecture. *IEEE Trans. on Computers: special issue on Advances in High Performance Memory Systems*, 2001.

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

[21] G. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of Intl. Conf. on Supercomputing*, 2001.

[22] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of Intl. Symp. on High Performance Computer Architecture*, 2002.

[23] D. Thiebaut, H. Stone, and J. Wolf. Footprints in the cache. *ACM Trans. on Computer Systems*, 5(4), Nov. 1987.

[24] X. Vera and J. Xue. Let's Study Whole-Program Cache Behaviour Analytically. In *Proc. of Intl. Symp. on High Performance Computer Architecture*, 2002.

[25] D. Vianney. Hyper-threading speeds linux: Multiprocessor performance on a single processor. *IBM DeveloperWorks*, 2003.

[26] 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 *Supercomputing*, 1997.