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}@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 causes potential performance problems that the Operating System (OS) must be aware of and avoid, such as suboptimal throughput, cache thrashing, and thread starvation for threads that fail to occupy sufficient cache space to make good progress. Despite the performance problems caused by cache sharing, currently there is no performance model that allows us to investigate the impact of cache sharing extensively.

We propose a performance model that predicts the impact of cache sharing on co-scheduled threads. The input to our model is the isolated L2 *circular sequence profile* of each thread, which can be easily obtained on-line or off-line. The output of the model is the extra number of L2 cache misses of each thread that shares the cache. We validate the model against a cycle-accurate simulation that implements a dual-core CMP architecture, on fourteen pairs of SPEC benchmarks. The model's average error is only 3.8%. Finally, to demonstrate that the model provides a valuable and practical tool through which we can study the impact of cache sharing extensively, we present a case study on synthetic profiles.

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]. An L2 cache that is shared by co-scheduled threads running on different cores allows high cache utilization and low cache fragmentation. However, as will be demonstrated in this paper, the impact of the contention of the shared cache is not uniform across the threads that share it. Some threads are slowed down significantly, while others are not impacted much. This causes potential performance problems that the Operating System (OS) must be aware of and avoid, such as suboptimal throughput, cache thrashing, and thread starvation for threads that fail to occupy sufficient cache space to make good progress.

To illustrate the problem, Figure 1 shows the number of cache misses per instruction (1a) and IPC (1b) for *mcf* when it is run 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* is run together with *mst* or *gzip*, *mcf* does not suffer from additional misses compared to when it runs alone. However, if it is run together with *art* or *swim*, its number of misses per instruction increases roughly twofold. The IPC figure shows almost the same trend. The IPC of *mcf* when it runs together with *art* and *gzip* is roughly a half of when it runs alone [1].

Despite the performance problems caused by cache sharing, there is no existing model that allows us to investigate the impact of cache sharing extensively. Previous performance prediction models only predict the number of cache misses in a uniprocessor system [3, 5, 6, 7, 11, 19, 20], or predict cache contention on a single processor time-shared system [1, 16, 18], 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 aims to go beyond past studies by presenting an accurate and tractable model that predicts the impact of cache contention between threads that simultaneously share the L2 cache on a Chip Multi-Processor (CMP) architecture. The input to our model is the isolated L2 cache stack distance and *circular sequence* profiling of each thread, which can be easily obtained on-line or off-line [4]. The output of the model is the extra

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

[1] Note that *mcf*'s IPC is reduced by roughly 25% when it is run with *mst*, even though its number of miss per instruction has not changed much. This is due to contention in other shared resources such as the memory bus. We found that the contention in the L2 cache space is the most important factors influencing the IPC of an application. In addition, other factors such as the degree of bus and bank contention are directly impacted by the number of L2 cache misses due to L2 cache contention/sharing. Thus, the focus of this paper is modeling L2 cache space contention.

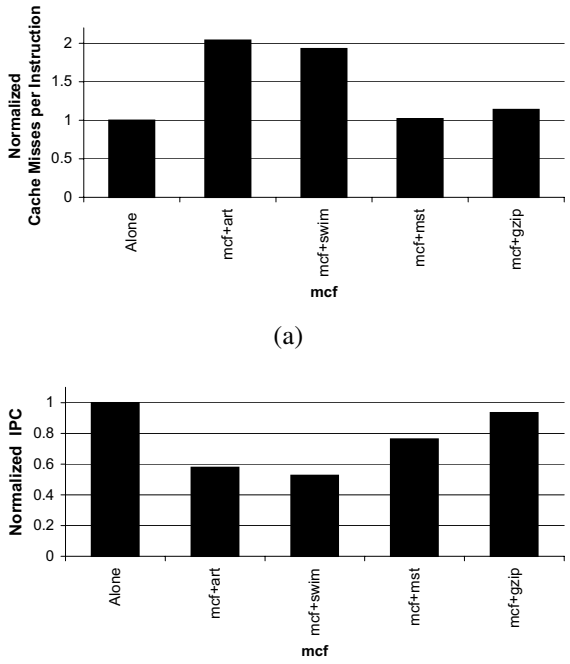

Figure 1: The number of L2 cache misses per instruction (a), and the IPC (b), for *mcf* when it is run alone compared to when it is co-scheduled with another thread. The L2 cache is 512KB, 8-way associative, and has 64-byte line size.

number of L2 cache misses of each thread that shares the cache.

The proposed model is based on inductive probability (*Prob*). We validate the model 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 3.8%. It correctly identifies all cases where the number of cache misses of a benchmark increases significantly under cache sharing; compared to if the benchmark runs alone. Finally, the model provides a valuable and practical tool through which we can study the impact of cache sharing extensively. 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 paper is organized as follows. Section 2 presents the prediction model. Section 3 details the validation setup for our model. Section 4 presents and discusses the model validation results. Finally, Section 5 summarizes the findings.

2. Cache Miss Prediction Model

This section will present the proposed prediction model. It discusses assumptions used by the model (Section 2.1), discusses an existing stack distance profiling method (Section 2.2), and presents the inductive probability model (Section 2.3).

2.1. Assumptions

We assume that each thread's temporal behavior can be captured by a single stack distance 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 stack distance profiles that represent different program phases may further improve the prediction accuracy, at the expense of extra complexity, e.g. [13]. This is beyond the scope of this paper.

It is also assumed that a stack distance profile of a thread is the same with or without sharing the cache with other threads. This assumption ignores the impact of 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 reuse frequency and stack distance profile at the L2 cache level. In the evaluation (Section 4), we test the effect of this assumption by relaxing the inclusion property, and found that the effect results in only a very small inaccuracy (0.6%).

Co-scheduled threads are assumed not to share any address space. The assumption is reasonable because if the co-scheduled threads are from different applications, they typically do not share much data. If the co-scheduled threads are from a parallel program, the threads may share a large amount of data. However, the threads are likely to have similar characteristics, making the cache sharing prediction trivial because the cache is likely to be equally divided by the threads and each thread is likely to be impacted the same way. Consequently, we ignore this case.

Finally, the L2 cache is assumed to use 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. Stack Distance Profiling

The input to the model is the isolated L2 cache *stack distance profile* of each thread without the impact of cache sharing. Stack distance profiling captures the temporal reuse behavior of an application in a fully- or set-associative cache [12, 3, 11, 14]. It is also referred to as marginal gain counters [16, 17]. For an A-way asso-

ciative 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. If it is a cache miss, the line is not found in the LRU stack, resulting in incrementing the miss counter $C_{>A}$. Stack distance profile can easily be obtained statically by the compiler [3], by simulation, or by running the thread alone in the system [17].

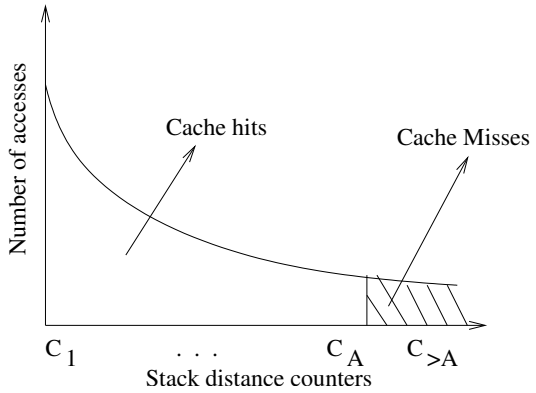

Figure 2: Illustration of a stack distance profile

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 monotonically decreasing values, as shown in Figure 2.

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 counter by the number of processor cycles in which the profile is collected (i.e., $Cf_i = \frac{C_i}{CPUcycle}$). Furthermore, we call $Cf_{>A}$ 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 call the sum of miss and reuse frequency as *access frequency* (Af).

2.3. Inductive Probability Model

In the inductive probability model (*Prob*), for each possible access sequence from a thread, we consider all possible interleaving of accesses from another thread, and compute the probability of such interleaving resulting in an additional cache misses. We also have two heuristics-based models that are covered in [4] but will not be discussed in this paper. They are simpler, but are not as accurate. Before we explain the model, it is useful to define several terms.

Definition 2.1 *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 block addresses by thread X, where all the accesses* map to the same cache set.

Definition 2.2 *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 block 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. This is illustrated in Figure 3. In the figure, there are eight accesses to five different block addresses that map to a 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 ($cseq(1,2)$).

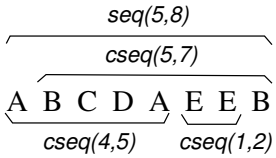

Figure 3: Illustration of the relationship between a sequence and circular sequences.

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* [2]. To achieve that, it is important to consider the following property.

Property 2.1 *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. 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

[2] Note that, there are some addresses that are accessed only once. They do not form circular sequences. Their accesses result in cache miss when run alone, and remain so under cache sharing. Therefore, we can safely ignore this case, and only consider the probability of a cache hit becoming a cache miss under cache sharing.

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 2.1 *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 $d_X(\leq A)$ appear it increments C_{d_X}. If $N(cseq_X(d_X, *))$ denotes number of occurrences of circular sequence $cseq_X(d_X, *)$, we have C_{d_X} = $N(cseq_X(d_X, *))$, we therefore have the following corollary.

Corollary 2.2 *The probability of occurrences of circular sequences $cseq_X(d_X, *)$ in total number of accesses from thread X is $P(cseq_X(d_X, *)) = \frac{C_{d_X}}{totAccess_X}$, 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 4 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.

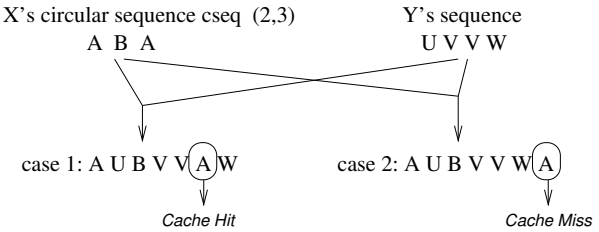

Figure 4: Illustration of how intervening accesses from another thread determines whether the last access of a circular sequence will be a cache hit or a miss. Capital letters in a sequence represent line addresses. We assume a 4-way associative cache and all accesses are to a single cache set.

Corollary 2.3 *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 ($seq_Y(d_Y, n_Y)$) 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 $d_X + d_Y \leq A$.* [3]

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 occurrences of a sequence $seq_Y(d_Y, n_Y)$ 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 expected values: $\overline{n_X}$ and $E(n_Y)$, respectively. So, if $P_{miss}(cseq_X(d_X, \overline{n_X}))$ denotes the probability of the last access of X's circular sequence becoming a cache miss, the following equation states corollary 2.3 in a formal expression:

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

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

[3] 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.

1. For each possible value of d_X, compute the weighted average of n_X ($\overline{n_X}$) by considering the distribution of $cseq_X(d_X, n_X)$. 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: $T(cseq_X(d_X, \overline{n_X}))$.
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, we use $seq_Y(d_Y, E(n_Y))$ instead of $seq_Y(d_Y, n_Y)$
4. For each possible value of d_Y, compute the probability $P(seq_Y(d_Y, E(n_Y)))$. Then, compute the probability of the last access of X's circular sequence becoming a cache miss by: $P_{miss}(cseq_X(d_X, \overline{n_X})) = \sum_{d_Y=A-d_X+1}^{E(n_Y)} P(seq_Y(d_Y, E(n_Y)))$
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.

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

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

$$\overline{n_X} = \frac{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X)) \times n_X}{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X, n_X))} \quad (2)$$

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 a simple hardware support described in [4]. We found that by considering n_X only up to 128, we get almost the same accuracy as considering larger n_X values, because typically there are not many circular sequences with really large n_X.

2.3.2. 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):

$$T(cseq_X(d_X, \overline{n_X})) = \frac{\overline{n_X}}{Af_X} \quad (3)$$

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 (Af_Y):

$$E(n_Y) = Af_Y \times T(cseq_X(d_X, \overline{n_X})) \quad (4)$$

2.3.3. 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 $P(seq_Y(d_Y, E(n_Y)))$. To compute $P(seq(d,n))$, we will use inductive probability function.

To compute $P(seq(d,n))$, let us look into the sequence in detail. The sequence indicates that there are n accesses to d distinct addresses. Suppose that the addresses that have been accessed so far include $\{Y_1, Y_2, \ldots, Y_d\}$. To incorporate the next access into the sequence, we analyze the address of the next access (Y_i). If $Y_i \in \{Y_1, Y_2, \ldots, Y_d\}$, the next address has been seen before in the sequence, and therefore the number of distinct addresses in the sequence is unchanged. The resulting new sequence is $seq(d, n+1)$. It means that there is a circular sequence that begins with Y_i and ends with Y_i, where the distinct addresses between them may vary from 0 (if Y_i is the same as the most recently seen address, i.e. $Y_i = Y_d$), to d (if Y_i is the same as the least recently seen address, i.e. $Y_i = Y_1$). Therefore, the probability of $Y_i \in \{Y_1, Y_2, \ldots, Y_d\}$ is equal to $\sum_{i=1}^{d} P(cseq(i,*))$, or denoted as $P(d^-)$ for simplicity. If, however, the address Y_i has not been seen before, i.e. $Y_i \notin \{Y_1, Y_2, \ldots, Y_d\}$, the probability for this case is equal to $1 - P(d^-)$, simply denoted as $P(d^+)$. In this case, the resulting new sequence will be $seq(d+1, n+1)$ because now we have one more access and one more distinct address. Corollary 2.2 and Figure 5 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 $P(3^+)$.

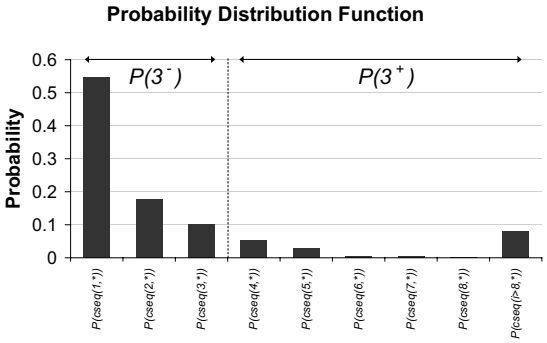

Figure 5: Example probability distribution function that shows $P(3^-)$ and $P(3^+)$. The function is computed by using the formula in Corollary 2.2.

To complete the formula, note that $seq(d,n)$ can occur only when the previous sequence is either $seq(d, n-1)$

or $seq(d-1, n-1)$. Therefore, we can express the probability $P(seq(d,n))$ with a recursive relation:

$$P(seq(d,n)) = P(d^-) \times P(seq(d,n-1)) + P((d-1)^+) \times P(seq(d-1,n-1)) \quad (5)$$

Equation 5 can be easily computed with recursion when we have the boundary values. The boundary values are:

$$P(seq(1,n)) = P(1^-) \times P(seq(1,n-1)) \quad (6)$$

$$P(seq(d,d)) = P((d-1)^+) \times P(seq(d-1,d-1)) \quad (7)$$

$$P(seq(1,1)) = 1 \quad (8)$$

Note that the last boundary value $P(seq(1,1)) = 1$ is true because the first address is always considered distinct.

2.3.4. 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. From corollary 2.3 and equation 1, we know that

$$P_{miss}(cseq_X(d_X, \overline{n_X})) = \sum_{d_Y = A - d_X + 1}^{E(n_Y)} P(seq_Y(d_Y, E(n_Y))) \quad (9)$$

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 ($C_{>A}$):

$$miss_X = C_{>A} + \sum_{d_X=1}^{A} P_{miss}(cseq_X(d_X, \overline{n_X})) \times C_{d_X} \quad (10)$$

3. Validation Methodology

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 [15]; and *mst* from Olden benchmark. Table 1 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 when one benchmark in the co-schedule finishes execution, the other is terminated, to make sure that the data collected only reflects inter-thread cache contention. These benchmarks are paired and co-scheduled. Fourteen benchmark pairs are evaluated.

Simulation Environment. The evaluation is performed using a detailed cycle-accurate execution-driven multiprocessor simulator. 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 [9].

Benchmark	Input Set	Miss Rate
art	test	66%
applu	test	36%
apsi	test	27%
equake	test	82%
gzip	test	3%
mcf	test	8%
perlbmk	reduced ref	46%
swim	test	76%
mst	1024 nodes	60%

Table 1: The applications used in our evaluation.

CMP
2 cores, each 4-issue dynamic. 3.2 GHz. Int, fp, ld/st FUs: 2, 2, 2 Pending ld, st: 8, 8. Branch penalty: 17 cycles Re-order buffer size: 192
MEMORY
L1 Inst (private): WB, 32 KB, 4 way, 64-B line, RT: 3 cycles, LRU L1 data (private): WB, 32 KB, 4 way, 64-B line, RT: 3 cycles, LRU L2 data (shared): WB, 512 KB, 8 way, 64-B line, RT: 14 cycles, LRU L2 indexing: prime modulo with 1021 number of sets RT memory latency: 407 cycles Memory bus: split-transaction, 8 B, 800 MHz, 6.4 GB/sec peak

Table 2: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*.

Co-scheduling. Benchmark pairs are 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 due to sharing the L2 cache. To obtain accurate stack distance profiles, for the shorter thread, the profile is collected for its entire execution in without cache sharing. But for the longer thread, the profile is collected until the number of instructions executed reaches the number of instructions executed in the co-schedule.

4. Evaluation and Validation

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

4.1. Impact of Cache Sharing

Figure 6 shows the impact of cache sharing on IPC of each benchmark in a co-schedule. Each bar has a black and a white sections. The full height of a bar (black + white sections) represents the IPC of the benchmark when it runs alone in the CMP, without sharing the L2 cache with another benchmark. The black section rep-

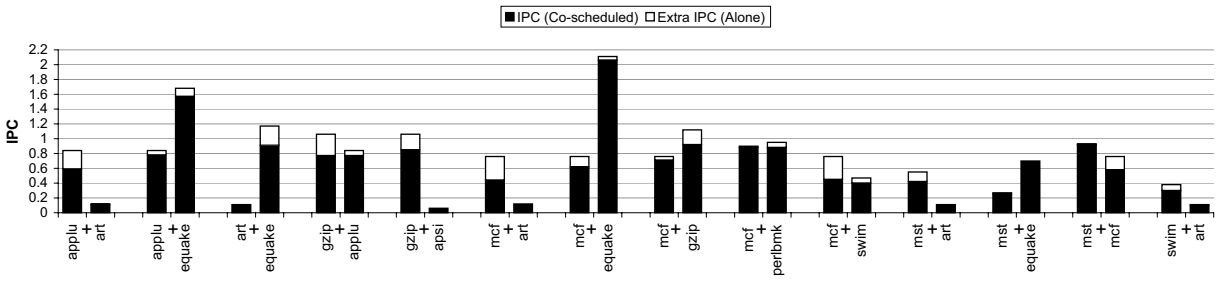

Figure 6: The impact of cache sharing on each co-scheduled thread's IPC.

resents the IPC of the benchmark when it is co-schedule with another benchmark. Therefore, the white section represents the reduction in IPC of the benchmark due to 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. In most co-schedules, the IPC reduction is usually highly different for the two benchmarks in each co-schedule. For example, *gzip* and *mcf* are slowed down by a lot more compared to their co-scheduled benchmarks: *gzip*'s IPC is reduced by between 20-28% in *gzip+applu*, *gzip+apsi*, and *gzip+mcf*; while *mcf*'s IPC is reduced by 40% in *mcf+art* and 45% in *mcf+swim*). However, sometimes *mcf* is not slowed down much, such as in *mcf+gzip*.

Another observation is that the benchmarks that do not suffer much slowdown under cache sharing are usually the ones that have a very high IPC (*equake* in *applu+equake* and *mcf+equake*) or a very low IPC (*art*) [4]. This is because for the duration of the co-schedule with *applu* and *mcf*, *equake* has a small working set that easily fits in the L2 cache. *art*, however, has a very large working set that overflows the L2 cache, even when it runs alone. Co-scheduling *art* with other benchmarks do not increase the number of L2 cache misses, since it already overflows the L2 cache. Benchmarks that have medium IPC values, such as *applu*, *equake* in *art+equake* and *mst+equake*, *gzip*, *mcf*, and *mst* are the most vulnerable to IPC reduction because they have a large working set that partially fits in the L2 cache. Sharing the cache with another benchmark decreases their effective cache space and displaces some parts of their working set from the L2 cache. Therefore, the temporal behavior of each application determines the impact of cache sharing it suffers. However, note that the observation so far is neither systematic nor precise. In order to really understand why some threads are vulnerable to large IPC reduction due to cache sharing, we present a case study in Section 4.3.

[4]Note that the difference of *equake*'s IPCs in *applu+equake* and *mcf+equake* is due to the difference in the duration of the two co-schedules. This effect also applies to all the benchmarks

4.2. Model Validation

Table 3 shows the validation results for the fourteen co-schedules that we evaluate. The first numeric column shows 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 the simulator. The next column presents the prediction errors of the inductive probability model (*Prob*). The errors are computed as the number of L2 cache misses predicted by the model minus that collected by the simulator, divided by the L2 cache misses collected by the simulator. 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 three rows in the table summarize the errors. They present the minimum, maximum, and the average 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. In five co-schedules, one of their benchmarks suffer from 65% or more extra cache misses: *gzip+applu* (240% extra misses in *gzip*), *gzip+apsi* (65% extra misses in *gzip*), *mcf+art* (104% extra misses in *mcf*), *mcf+gzip* (208% extra misses in *gzip*), and *mcf+swim* (92% extra misses in *mcf*).

The table shows that overall, *Prob* is very accurate. Its average error is only 3.8%, while its maximum absolute error is only 22%. Analyzing the errors for different co-schedules, *Prob*'s prediction errors are larger than 10% *only when* a benchmark suffers a very large increase in cache misses, such as *gzip* in *gzip+applu* (-22% error, 240% extra cache misses), and *gzip* in *mcf+gzip* (18% error, 208% extra cache misses). It is less critical to predict those cases very accurately, because the model still correctly identifies the cases where the number of cache misses increases significantly. Elsewhere, *Prob* is able to achieve a very high accuracy, even when in some cases

Co-schedule		Extra L2 Cache Misses Due to Sharing	L2 Cache Miss Prediction Error (E_j) of *Prob*
applu	applu	21%	3%
+art	art	0%	0%
applu	applu	8%	1%
+equake	equake	20%	8%
art	art	0%	0%
+equake	equake	16%	0%
gzip	gzip	240%	-22%
+applu	applu	13%	1%
gzip	gzip	65%	-9%
+apsi	apsi	4%	-4%
mcf	mcf	104%	-5%
+art	art	0%	0%
mcf	mcf	6%	-3%
+equake	equake	8%	8%
mcf	mcf	14%	3%
+gzip	gzip	208%	18%
mcf	mcf	5%	-1%
+perlbmk	perlbmk	29%	-4%
mcf	mcf	92%	-7%
+swim	swim	0%	0%
mst	mst	15%	1%
+art	art	0%	0%
mst	mst	21%	5%
+equake	equake	3%	0%
mst	mst	0%	4%
+mcf	mcf	2%	1%
swim	swim	0%	0%
+art	art	0%	0%
Minimum Absolute Error ($min(\|E_j\|)$)			0%
Maximum Absolute Error ($max(\|E_j\|)$)			22%
Average Absolute Error ($avg(\|E_j\|)$)			**3.8%**

Table 3: Validation Results

there is a large number of extra cache misses. For example, for *mcf* in *mcf+art*, the error is only -5%; and for *mcf* in *mcf+swim*, the error is only -7%.

Prob's remaining inaccuracy may be caused by three simplifying assumptions. First, the model ignores the impact of cache inclusion property. 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 model assumes to be unchanged. To find out how much accuracy is lost due to this assumption, we simulate a non-inclusive L2 cache and rerun the *Prob* model, and found that the average error decreases slightly to 3.2%. Secondly, we assumed 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). Finally, we assumed that the number of accesses from an interfering thread Y can be represented accurately by its expected value ($E(n_Y)$ in Equation 4). Relaxing these assumptions require treating n_X and n_Y as random variables, which significantly complicates the *Prob* model.

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

The case study evaluates how temporal reuse behavior affects the impact of cache sharing, by generating synthetic stack distance and circular sequence 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*. The base thread is the thread that we want to investigate.

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. Note that this is in general a good 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 one factor of the interfering thread that affects the impact of cache sharing.

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

Figure 7 show the result of the three experiments, where we vary the reuse frequency (a), miss frequency (b), and common ratio (c) of the interfering thread. The x-axes of subfigure (a) and (b) show the different multiplying factor values (k) for the interfering thread. The x-axes of subfigure (c) shows the different values of the common ratio of the interfering thread (r'). The y-axes of all the subfigures show the number of extra cache misses over the case where the base thread runs alone, divided by the amplitude Z. Each subfigure has multiple lines, where different lines in each subfigure represent a base thread with different common ratio values (r).

Figure 7a shows how reuse frequency determines the cache sharing impact. The figure shows that a base thread with a higher common ratio (i.e., flatter stack distance) not only suffers more extra cache misses, but the misses also increase more rapidly with a higher reuse frequency in the interfering thread (i.e., along the x-axes). Figure 7b

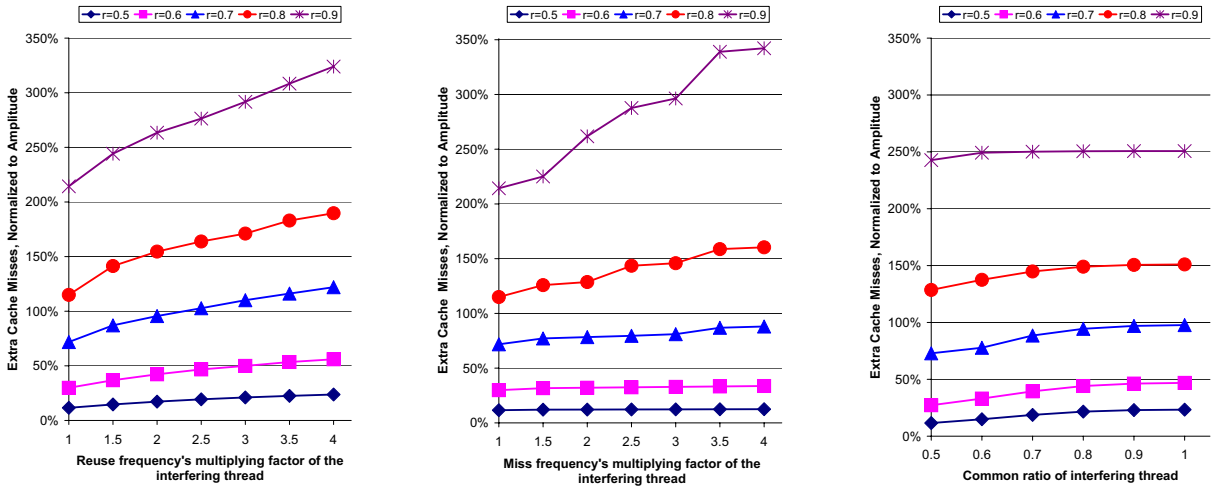

Figure 7: Relationship between temporal reuse behavior and cache sharing impact. The figure shows the base thread's number of extra cache misses under cache sharing, when we vary the interfering thread's reuse frequency (a), miss frequency (b), and common ratio (c). Each line in the figure is the base thread's number of extra cache misses for a particular common ratio value.

shows that the impact of the higher miss frequency of the interfering thread is similar to higher reuse frequency of the interfering thread in Figure 7a.

Figure 7c shows a different trend. Although a base thread with a higher common ratio (i.e., flatter stack distance) suffers more extra cache misses, the cache misses increase less rapidly along the x-axes, where the interfering thread's stack distance profile becomes flatter.

Overall, the figure shows that the impact of cache sharing is mainly determined by the base thread's stack distance profile shape, and the interfering thread's reuse and miss frequencies. This explains why in Table 3 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 cache misses in *gzip+applu* compared to in *gzip+apsi*.

5. Conclusions and Future Work

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. To avoid these problems, ideally the Operating System (OS) thread scheduler should be able to predict which co-schedules are or are not likely to cause cache thrashing and thread starvation, allowing it to choose co-schedules judiciously.

We propose a performance model that predicts the impact of cache sharing on co-scheduled threads. The input to our model 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 model is the extra number of L2 cache misses of each thread that shares the cache. We validate the model against a cycle-accurate simulation that implements a dual-core CMP architecture, on fourteen pairs of SPEC benchmarks. The model's average error is only 3.8%. We have also identified some possible sources of inaccuracy that will be investigated in the near future to improve the model further.

Finally, the model provides a valuable and practical tool through which we can study the impact of cache sharing extensively. We present a case study to demonstrate how different temporal reuse behaviors in applications influence the impact of cache sharing suffered by them. The significance of the case study in Section 4.3 is not only in understanding the cache sharing impact, but possibly also in parameterizing applications based on their stack distance profiles so that the OS scheduler can better avoid co-schedules that result in pathological perfor-

mance behavior. Once a good co-schedule is constructed, a fair caching scheme may be applied to improve fairness among of the threads and possibly improve their combined throughput [10].

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. In *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] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proc. of the Intl. Symp. on High Performance Computer Architecture*, To appear, 2005.

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

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

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

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

[20] 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.