Predicting Cache Space Contention in Utility Computing Servers

Yan Solihin, Fei Guo, and Seongbeom Kim
Department of Electrical and Computer Engineering
North Carolina State University
{solihin,guo,skim16}@ece.ncsu.edu
http://www.cesr.ncsu.edu/solihin

Abstract

The need to provide performance guarantee in high performance servers has long been neglected. Providing performance guarantee in current and future servers is difficult because fine-grain resources, such as on-chip caches, are shared by multiple processors or thread contexts. Although inter-thread cache sharing generally improves the overall throughput of the system, the impact of cache contention on the threads that share it is highly non-uniform: 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. Clearly, this situation is not desirable when performance guarantee needs to be provided, such as in utility computing servers.

Unfortunately, there is no existing model that allows extensive investigation of the impact of cache sharing. To allow such a study, we propose an inductive probability model to predict the impact of cache sharing on co-scheduled threads. The input to the 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 number of extra L2 cache misses for each thread due to cache sharing. We validate the model against a cycle-accurate simulation that implements a dual-core Chip Multi-Processor (CMP architecture), on fourteen pairs of mostly SPEC benchmarks. The model achieves an average error of only 3.9%.

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.

![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.](image-url)
that detect but not prevent the problems, and apply inter-thread cache partitioning schemes to improve fairness [11] or throughput [17, 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 an analytical model to predict the impact of cache sharing.

Past performance prediction models only predict the number of cache misses in a uniprocessor system [3, 5, 6, 7, 12, 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 goes beyond past studies and presents a tractable, analytical inductive probability model (Prob) that predicts the impact of cache space contention between threads that simultaneously share the L2 cache on a Chip Multi-Processor (CMP) architecture. The input to the model is the isolated L2 cache circular sequence profiling of each thread, which can be easily obtained on-line or off-line. The output of the model is the number of extra L2 cache misses of each thread that shares the cache. 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 3.9%.

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

2 Cache Miss Prediction Model

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. 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.

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 many other replacement policies as well.

2.2 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 [13, 3, 12, 14], and is sometimes also referred to as marginal gain counters [16, 17]. 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
A stack distance profile can be easily obtained statically by the compiler [3], by simulation, or by running the thread alone in the system [17].

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., \( CF_i = \frac{C_i}{CPU\text{-}cycles} \)). 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).

2.3 Inductive Probability (Prob) Model

The Prob model computes the probability of each cache hit turning into a cache miss, for every possible access interleaving by an interfering thread. 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 2. In the figure, there are eight accesses to five different line addresses that map to a cache set. In it, there are three circular sequences that are overlapping: one that starts and ends with address \( A \) (cseq\((4,5)\)), another one that starts and ends with address \( B \) (cseq\((5,7)\)), and another one that starts and ends with address \( E \) (cseq\((1,2)\)).

![Figure 2: Illustration of the relationship between a sequence and circular sequences.](image)

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. 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 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.
$N(cseq_X(d_X, *))$ 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 $P(cseq_X(d_X, *)) = \frac{C_{d_X}}{\text{totAccess}_X}$, where $\text{totAccess}_X$ 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 3 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 interleave between accesses from thread $X$ and $Y$ are possible. The figure shows two of the access interleave 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 $d_X + d_Y \leq A$.  

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

Note that we now deal with a probability computation with four random variables ($d_X, n_X, d_Y$, and $n_Y$). To simplify the computation, we represent $n_X$ and $n_Y$ by their expected values: $\bar{n_X}$ and $E(n_Y)$, respectively. Hence, Corollary 3 can be formally stated as:

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

(1)

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., $\bar{n_X}$) by considering the distribution of $cseq_X(d_X, n_X)$. This requires a circular sequence profiling [4]. Then, we use $cseq_X(d_X, \bar{n_X})$ instead of $cseq_X(d_X, n_X)$.

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

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

---

2For 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.
Then, compute the probability of the last access of $X$’s circular sequence becoming a cache miss by using Equation 1.

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 $\pi_X$

$\pi_X$ is computed by taking its average over all possible values of $n_X$:

$$\pi_X = \frac{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X,n_X)) \times n_X}{\sum_{n_X=d_X+1}^{\infty} N(cseq_X(d_X,n_X))}$$  \hspace{1cm} (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 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.

2.3.2 Step 2 and 3: Computing $T(cseq_X(d_X,\pi_X))$ 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 $X$ ($Af_X$):

$$T(cseq_X(d_X,\pi_X)) = \frac{\pi_X}{Af_X}$$  \hspace{1cm} (3)

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

$$E(n_Y) = Af_Y \times T(cseq_X(d_X,\pi_X))$$  \hspace{1cm} (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 $P(seq_Y(d_Y,E(n_Y)))$. 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. $P(seq(d,n)) = \begin{cases} 1 & \text{if } n = d = 1 \\ P((d-1)^+) \times P(seq(d-1,d-1)) & \text{if } n = d > 1 \\ P(1^-) \times P(seq(1,n-1)) & \text{if } n > d = 1 \\ P(d^-) \times P(seq(d,n-1)) + P((d-1)^+) \times P(seq(d-1,n-1)) & \text{if } n > d > 1 \end{cases}$

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 $P((d-1)^+)$.

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($1,n-1$) 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, $P(seq(d,n)) = \sum_{i=1}^{d} P(cseq(i, *))$ is calculated as $C_{d,A} \sum_{i=1}^{d} C_i$.
\(P((d - 1)^+) \times P(seq(d - 1, d - 1))\) follows from Case 1.

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

Corollary 2 and Figure 4 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 \(P(3^+)\).

### 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 \(d_Y\). We can then compute \(P_{\text{miss}}(cseq_X(d_X, \pi_X))\) using Equation 1. 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 (\(C_{\geq A}\)):

\[
\text{miss}_X = C_{> A} + \sum_{d_X = 1}^{A} P_{\text{miss}}(cseq_X(d_X, \pi_X)) \times C_{d_X} \quad (5)
\]

## 3 Validation Methodology

**Simulation Environment.** The evaluation is performed using a detailed CMP architecture simulator based on SESC, a cycle-accurate execution-driven simulator [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 1 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 1: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. \(RT\) stands for round-trip from the processor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP</strong></td>
<td></td>
</tr>
<tr>
<td>Cores</td>
<td>2</td>
</tr>
<tr>
<td>FUs</td>
<td>3, 2, 2</td>
</tr>
<tr>
<td>Int, fp, ld/st</td>
<td></td>
</tr>
<tr>
<td>Branch penalty</td>
<td>13 cycles</td>
</tr>
<tr>
<td>Re-order buffer size</td>
<td>152</td>
</tr>
<tr>
<td><strong>MEMORY</strong></td>
<td></td>
</tr>
<tr>
<td>L1 Inst, Data (private):</td>
<td>each WB, 32 KB, 4 way, 64-B line, RT: 2 cycles, LRU replacement</td>
</tr>
<tr>
<td>L2 Inst, Data (shared):</td>
<td>WB, 512 KB, 8 way, 64-B line, RT: 12 cycles, LRU replacement, prime modulo indexed, inclusive.</td>
</tr>
<tr>
<td>RT memory latency</td>
<td>362 cycles</td>
</tr>
<tr>
<td>Memory bus:</td>
<td>split-transaction, 8 B, 800 MHz, 6.4 GB/sec peak</td>
</tr>
</tbody>
</table>

**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, perlbench and swim from the SPEC2K benchmark suite [15]; and mst from Olden benchmark suite. Table 2 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.
4 Validation

Table 3 shows the validation results for the fourteen co-schedules that we evaluate. The first numeric column shows the number of 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 twice compared to when the benchmark runs alone). The cache misses are collected using simulation. The last column presents the prediction error of the model. 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.

The impact of cache sharing is significant. 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).

Analyzing the errors for different co-schedules, Prob’s prediction errors are generally small. They 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, but the error is only -9%.

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 (nX in Equation 2). We also assumed that the number of accesses from an interfering thread Y can be represented accurately by its expected value (E(nY) in Equation 4). In addition, the model rounds down E(nY) to the nearest integer. Relaxing these assumptions requires treating nX and nY as random variables, which significantly

<table>
<thead>
<tr>
<th>Co-schedule</th>
<th>Extra L2 Cache Misses Due to Sharing</th>
<th>L2 Cache Miss Pred Error Ej of Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>29%</td>
<td>2%</td>
</tr>
<tr>
<td>+art</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>applu</td>
<td>16%</td>
<td>1%</td>
</tr>
<tr>
<td>+eqquake</td>
<td>19%</td>
<td>5%</td>
</tr>
<tr>
<td>art</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>+eqquake</td>
<td>43%</td>
<td>5%</td>
</tr>
<tr>
<td>gzip</td>
<td>243%</td>
<td>-25%</td>
</tr>
<tr>
<td>+applu</td>
<td>11%</td>
<td>2%</td>
</tr>
<tr>
<td>gzip</td>
<td>180%</td>
<td>-9%</td>
</tr>
<tr>
<td>+apsi</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mcf</td>
<td>296%</td>
<td>7%</td>
</tr>
<tr>
<td>+art</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mcf</td>
<td>11%</td>
<td>-3%</td>
</tr>
<tr>
<td>+eqquake</td>
<td>0%</td>
<td>5%</td>
</tr>
<tr>
<td>mcf</td>
<td>18%</td>
<td>7%</td>
</tr>
<tr>
<td>+gzip</td>
<td>102%</td>
<td>22%</td>
</tr>
<tr>
<td>mcf</td>
<td>8%</td>
<td>-3%</td>
</tr>
<tr>
<td>+perlbk</td>
<td>28%</td>
<td>2%</td>
</tr>
<tr>
<td>mcf</td>
<td>59%</td>
<td>-7%</td>
</tr>
<tr>
<td>+swim</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mcf</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td>+art</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>mcf</td>
<td>25%</td>
<td>3%</td>
</tr>
<tr>
<td>+eqquake</td>
<td>3%</td>
<td>0%</td>
</tr>
<tr>
<td>mcf</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>+mcf</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>swim</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>+art</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Min. Abs. Error: min(|Ej|) = 0%
Max. Abs. Error: max(|Ej|) = 25%
Mean Abs. Error: \( \sum_{j} |E_j| / n \) = 3.9%
Geom-Mean Abs. Error: \((\Pi(1 + |E_j|))^1/n - 1\) = 3.7%
complicates the Prob model.

5 Conclusions

This work has studied the impact of inter-thread cache contention on a Chip Multi-Processor (CMP) architecture. We have proposed an analytical model to predict the impact of cache sharing on co-scheduled threads. The input to our model is the isolated L2 cache 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 validated the model against a cycle-accurate simulation that implements a dual-core CMP architecture and found that the model produces very accurate prediction. Future work includes using the Prob model as a foundation for job admission and scheduling in a utility computing servers.

References


