L2 Cache Modeling for Scientific Applications on Chip Multi-Processors [*]

Fengguang Song
Dept. of Computer Science
University of Tennessee
Knoxville, TN
song@cs.utk.edu

Shirley Moore
Dept. of Computer Science
University of Tennessee
Knoxville, TN
shirley@cs.utk.edu

Jack Dongarra
Dept. of Computer Science
University of Tennessee
Knoxville, TN
Oak Ridge National Laboratory
Oak Ridge, TN
dongarra@cs.utk.edu

Abstract

It is critical to provide high performance for scientific applications running on Chip Multi-Processors (CMP). A CMP architecture often comprises a shared L2 cache and lower-level storages. The shared L2 cache can reduce the number of cache misses if the data are accessed in common by several threads, but it can also lead to performance degradation due to resource contention. Sometimes running threads on all cores can cause severe contention and increase the number of cache misses greatly. To investigate how the performance of a thread varies when running it concurrently with other threads on the remaining cores, we develop an analytical model to predict the number of misses on the shared L2 cache. In particular, we apply the model to thread-parallel numerical programs. We assume that all the threads compute homogeneous tasks and share a fully associative L2 cache. We use circular sequence profiling and stack processing techniques to analyze the L2 cache trace to predict the number of compulsory cache misses, capacity cache misses on shared data, and capacity cache misses on private data, respectively. Our method is able to predict the L2 cache performance for threads that have a global shared address space. For scientific applications, threads often have overlapping memory footprints. We use a cycle accurate simulator to validate the model with three scientific programs: dense matrix multiplication, blocked dense matrix multiplication, and sparse matrix-vector product. The average relative errors for the three experiments are 8.01%, 1.85%, and 2.41%, respectively.

Keywords: architecture, chip multi-processor, cache performance modeling, multi-threaded programming

[*]This material is based upon work supported by the National Science Foundation under grant No. 0444363 and by the Department of Energy Office of Science under grant No. DE-FC02-06ER25761.

1 Introduction

Cache performance plays an important role in software performance. With the increasing gap between memory and CPU speeds, it is essential to utilize the cache to its full potential. In recent years, Chip Multi-Processing (CMP) architectures have been developed to enhance performance and power efficiency through the exploitation of both instruction-level and thread-level parallelism. For instance, the IBM Power5 processor enables two SMT threads to execute on each of its two cores and four chips to be interconnected to form an eight-core module [11]. Intel Montecito, Woodcrest, and AMD AMD64 processors all support dual-cores [9]. Sun also shipped eight-core 32-way Niagara processors in 2006 [7].

In these architectures, some share an on-chip L2 cache among cores and others own private L1/L2 caches. As described in the prior work by Fedorova [5], an L2 cache miss penalty can be as high as 200-300 cycles while an L1 cache miss only costs a few cycles. Poor L2 cache behavior can dramatically increase the amount of off-chip communication and degrade the overall performance. Thus, our work is focused on modeling the behavior of the on-chip shared L2 cache. For multi-threaded programs, the shared L2 cache allows higher utilization of the L2 cache as a thread can reuse the same data loaded previously by another thread. Such reuse reduces power consumption and avoids duplicating hardware resources. However, parallel threads often interfere with each other and contend for accesses to the shared L2 cache, leading to suboptimal performance. We present an analytical model to predict the number of L2 cache misses for shared-memory scientific applications. By analyzing the L2 cache trace which is recorded when just a single thread is running, our model is able to predict the number of misses if we run the thread together with other threads on the remaining cores. We assume there is one thread on each core.

Considering the characteristics of thread-parallel programs from scientific computation, nearly all threads are homogeneous. That is, each thread works on the same task in parallel and has similar temporal behavior. Our model is an extension to Chandra's work [3]. The difference is that we use an offline approach to analyze the L2 cache trace and take into account the factor of memory sharing between threads. Being able to model the effect of shared memory accesses leads to a more powerful model that can predict the number of L2 cache misses for threads not only from distinct processes, but also from a single process.

The method presented in this paper classifies cache misses into three types: *compulsory misses*, *capacity misses on shared data*, and *capacity misses on private data*. The terms *shared* and *private* indicate whether the data are referenced by more than one thread (shared) or by a single thread (private). For instance, threads from different processes usually have disjoint memory accesses. We model the above three types of misses with three different methods: an average value for modeling compulsory misses, a probability method for modeling capacity misses on private data, and an effective cache space and a probability method for modeling capacity misses on shared data. Our goal is to model the L2 cache behavior by discovering the causes a cache hit developing into a cache miss as well as a cache miss turning into a cache hit.

The model is validated using the cycle-accurate simulator SESC [10]. Three scientific programs have been implemented to verify the model: matrix multiplication using three nested loops, blocked matrix multiplication, and sparse matrix-vector product taking as input sparse matrices from Matrix Market [2].

This paper is organized as follows: the next section introduces related work. Section 3 outlines the concepts and techniques used by the model. Section 4 describes in detail how we model the three miss types by different methods. In Section 5 we present the experimental results evaluating the model. Finally, Section 6 concludes the paper.

2 Related work

Agarwal [1] developed a cache model that combines measurement and analytical techniques to give miss rates for a given trace. The model is a function of a small number of factors that affect cache performance. It estimates cache performance for both a single process and multiple round-robin processes. Thiébaut [13] presents a model for cache-reload transients occurring in a multitasking system. The estimate provided by the model is dependent on the cache size and the footprints of the competing processes. Since both models only consider the process swapping effect, they are not suitable for modeling concurrent accesses to a shared cache from multiple processes or threads.

Mattson [8] describes a technique called *stack processing* to evaluate storage hierarchies. By one-pass scanning an address trace, the frequency of stack distances can be obtained to determine the miss rate function. The stack processing technique needs a memory trace and thus only a single program run is required. Suh [12] used a set of hardware counters (fully-associative counters, way-counters, and set-counters) to monitor the marginal gains in cache hits as the cache size is increased. These methods predict the miss rate as a function of cache size, but they require that address trace be fixed. Ding [4] measures program locality by *reuse distance* and presents a two-step strategy to maximize program locality. This strategy alleviates the pressure on insufficient memory bandwidth. Chandra [3] introduced an inductive probability model using circular sequences to predict cache interference from other threads. The probability model assumes that co-scheduled threads do not share any address space. However, there are many scientific applications that use the shared-memory programming model (e.g., OpenMP programs). Our research targets the new problem of cache modeling for shared-memory programs.

3 Methodology overview

We can use the stack processing technique to estimate the number of L2 cache misses given a fixed L2 cache trace. However, the L2 cache trace might be changed if we run a thread together with other threads due to the shared memory accesses. Therefore, we must be able to predict how a thread's trace is affected by the other threads (we call it "interference"). To realize the prediction, we employ the technique of circular sequence profiling. Note that we still need the stack processing technique to derive the number of cache misses from the estimated trace.

3.1 Stack processing technique

Gecsei introduces a technique called "stack processing" to evaluate storage hierarchies that use stack algorithms as a replacement policy [8]. A storage hierarchy consists of multiple levels of devices that are partitioned into *pages* or *blocks*. The input to the model is a *page trace* $x_1, x_2, \ldots, x_n$, where x_i is the page number accessed by the program. It is possible to apply the technique to any level of the storage hierarchy as long as there is a corresponding trace. We call the trace a *block trace* if we are examining caches.

Assume a fully-associative cache has $\mathcal{C}$ lines (or ways). It is easy to see that at any time t under LRU the cache contains the $\mathcal{C}$ most recently used lines. Even if we increase the cache size to $\mathcal{C}+1, \mathcal{C}+2, \ldots$, the set of $\mathcal{C}$ lines are still in the cache. This property is called the "inclusion property" and is formally defined in [8]. Because of the inclusion property, the content of the cache at any time t is able to be represented as an LRU stack

$$\mathcal{S}^{(t)} = \{s^{(t)}(1), s^{(t)}(2), \cdots, s^{(t)}(\mathcal{C})\}, \text{ where}$$
$$s^{(t)}(i) = Blocks^{(t)}(\mathcal{C} = i) - Blocks^{(t)}(\mathcal{C} = i - 1).$$

$Blocks(m)$ denotes the set of lines contained in a cache of size m. $s^{(t)}(i)$ is also known as "marginal gain" [12]. If a cache line x_t has been referenced before, the position $\triangle$ of that line counted from the top of the stack is called "stack distance". Let `counter(`$\triangle$`)` accumulate the number of times the stack distance $\triangle$ appears in the page trace. Such a set of counters forms a so-called *stack distance profile*. For instance, `counter(1)` counts the number of hits in the most recently used line, `counter(`$\mathcal{C}$`)` counts the number of hits in the least recently used line, and `counter(`$\mathcal{C}+1$`)` counts the number of cache misses.

Our model assumes a fully associative cache that uses the LRU algorithm and has no conflict misses. It has been shown that set-associative cache miss ratios are related to fully associative ones and a model using Bayes rule is able to make quite accurate predictions [6]. In addition, when the number of set-associativity is large, set-associative caches often have a miss rate comparable to fully associative caches.

A stack distance profile is sufficient to estimate the number of misses for a particular cache capacity. However, a page trace is prone to change because of other threads running on the remaining cores. Hence we must acquire more information to model the possible interferences from the other threads. The concept of *circular sequence profile* was introduced by Chandra [3] and has successfully modeled the interference effect from other processes in different address spaces. Note that we can deduce a stack distance profile from a circular sequence profile easily.

3.2 Circular sequence profile

A *circular sequence* is a sequence of cache lines $x_1, x_2, \cdots, x_n$ where $x_1 = x_n$, and x_1 does not appear anywhere in the middle of the sequence except for the beginning and the end positions [3]. It is possible that other circular sequences exist in the sequence if one cache line appears several times in the middle. For instance, the trace in Figure 1 contains five circular sequences. We use $CSEQ(d, n)$ to denote some set of circular sequences, in which each sequence is of length n and has d distinct cache lines:

$$CSEQ(d, n) = \{\alpha \text{ is a circular sequence} \mid \alpha \text{ accesses n lines and has d distinct lines}\}.$$

d and n define a different circular sequence set. In practice we use a counter to record the number of elements in

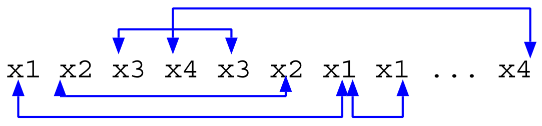

Figure 1. An example of a cache block trace containing five circular sequences

a nonempty $CSEQ(d, n)$. It is denoted as $|CSEQ(d, n)|$. Each $CSEQ(d, n)$ has a counter and the list of counters form a circular sequence profile.

We extend the SESC simulator to collect an L2 cache trace that consists of L2 lines sent from all processor cores. In the trace file, each cache line is written in the form of `PhysicalAddress:CoreId:VirtualAddress`. The second field `CoreId` helps keep track of a specific thread's trace, and the third field `VirtualAddress` is used to distinguish shared data accesses from private data accesses.

To obtain circular sequence profiles of each core, we use a scan process to analyze the L2 trace. Figure 2 shows the process's C++ program. Associative map `addr_map` records physical addresses and their indices in the trace, array `compulsory` counts the total number of compulsory misses for each core, and `cseq_shared`, `cseq_private` are circular sequence counters for shared and private data. The analysis process outputs not only compulsory misses for each core, but also circular sequence profiles for shared data $cseq^{shared}(d, n)$ and private data $cseq^{private}(d, n)$. Based on these three components, we are able to estimate the number of cache misses if running multiple threads simultaneously.

4 Modeling strategy

We use the most well-known "three Cs" model (compulsory, capacity, and conflict misses) to classify cache misses. For simplicity, we only consider fully associative caches. Our model takes as input a thread's circular sequence profile and estimates the number of misses if the thread had run together with other threads. Note that we always measure and predict the performance of the initial single thread. Since we don't consider simultaneous multi-threading (SMT) on processor cores and always have one thread per core, we interchangeably use "thread" and "core".

We analyze the L2 cache trace of a single-thread execution to estimate the number of L2 misses for a multiple-thread execution as follows: the scan process in Figure 2 creates a circular sequence profile, from which we can derive the number of cache misses:

$$misses = compulsory + \sum_{d>C} \sum_{n>d} |CSEQ(d, n)|.$$

```
map<paddr, pos> addr_map;
pos = 1;
while(not end of the file) {
  read into paddr, coreid, vaddr;
  if(addr_map[paddr] == 0) {
    addr_map[paddr] = pos;
    compulsory[coreid]++;
  }else {
   n = pos - addr_map[paddr] + 1;
   d = get_num_distinct
        (addr_map[paddr], pos-1);
   addr_map[paddr] = pos;
   if(is_shared(vaddr))
     cseq_shared[coreid][n][d]++;
   else
     cseq_private[coreid][n][d]++;
  }
  pos++;
}
```

Figure 2. The C++ program to scan the L2 cache trace to obtain circular sequence profiles and the number of compulsory misses for each processor core

If we run a thread together with other threads on different cores, the trace of that thread will be affected by references from the other threads. We divide L2 cache references into two types based on their addresses: references to shared data, and references to private data. Instead of using a single circular sequence profile $cseq(d, n)$, we introduce $cseq^{private}(d, n)$ and $cseq^{shared}(d, n)$ for each thread. For instance, consider two sequences: $ABCDA$ where A is shared and $ZBCDZ$ where Z is private. The first sequence increases the counter $cseq^{shared}(4, 5)$, while the second increases $cseq^{private}(4, 5)$.

Given a thread, different types of references are affected differently by the other threads. For instance, when a shared datum is accessed by two threads, the cache line previously loaded by one thread can save the other from reloading it. Therefore, (i) an original compulsory cache miss might become a hit. Another type is that (ii) the number of capacity misses on private data should increase because a previous hit may become a miss due to interferences from other threads. Finally, the prediction of capacity misses on shared data is much more complicated. (iii) A cache miss on shared data may become a hit because the other thread already loaded the data, meanwhile (iv) a cache hit on shared data may become a miss owing to other threads' interference.

In the following sections, we will describe our methods to predict the above three types of misses respectively. $Misses^{(co)}_{new}$ denotes the predicted number of compulsory misses, $Misses^{(pr)}_{new}$ denotes the predicted number of capacity misses on private data, and $Misses^{(sh)}_{new}$ denotes the number of capacity misses on shared data.

4.1 Modeling compulsory misses

An accurate method to determine how many compulsory misses become hits is dependent upon the relative speed of the concurrent threads and how much their working sets overlap. Given thread 0, thread 1, and shared data accesses b_1, b_2, b_3, b_4, thread 0 will have four compulsory misses if it is running alone. With thread 1 running, thread 0 misses might become fewer if thread 1 loads some of the data, or remain to be four if thread 1 always lags behind thread 0. It is hard to provide a precise prediction unless we know more detailed information.

Since we are concerned with homogeneous threads, it is reasonable to assume that the shared data are loaded evenly by the participating threads. This assumption has been validated by our experiments and most of the time the relative error for the compulsory miss estimate is less than 15%. Figure 3 shows an example that launches two threads to compute $C = A \times B$ using a block data distribution. Matrix B is shared by thread 0 and thread 1. From the perspective of thread 0, around half of its compulsory misses on matrix B may be loaded by thread 1.

We introduce $F^{(co)}_{m2h}$ to denote the fraction of a thread's compulsory misses that may become cache hits:

$$F^{(co)}_{m2h} = \frac{\text{Overlapped Blocks}}{TotalBlocks \times NumCores}.$$

The fraction of compulsory misses that remain to be compulsory misses $F^{(co)}_{miss}$ is as follows:

$$F^{(co)}_{miss} = 1 - F^{(co)}_{m2h}.$$

Thus the predicted number of compulsory misses if we run the thread together with other threads is expressed as:

$$Misses^{(co)}_{new} = Misses^{(co)}_{old} \times F^{(co)}_{miss},$$

where $Misses^{(co)}_{old}$ is the original number of compulsory misses when the thread is running alone.

4.2 Modeling capacity misses on private data

Any access to a private datum is either a miss or a hit. It is easy to see that every capacity miss on private data is still a cache miss regardless of whether the thread is running alone or with another thread. But a cache hit may become a miss because references of other threads will likely stretch out the circular sequence too much. Figure 4 illustrates how a cache hit could become a miss for a cache of size $\mathcal{C} = 4$.

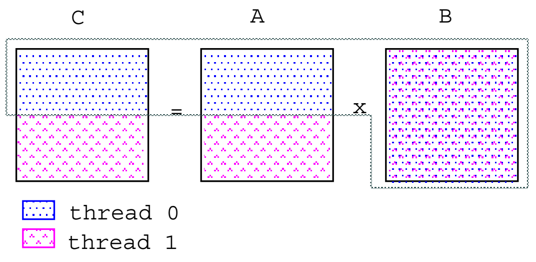

Figure 3. Two threads compute matrix multiplication of $C = A \times B$ using block data distribution. For thread 0, half of its compulsory misses on matrix B may be saved by data loading of thread 1 (i.e., $F^{(co)}_{m2h} = \frac{1}{(.5+.5+1)\times 2} = \frac{1}{4}$ of the number of the original compulsory misses).

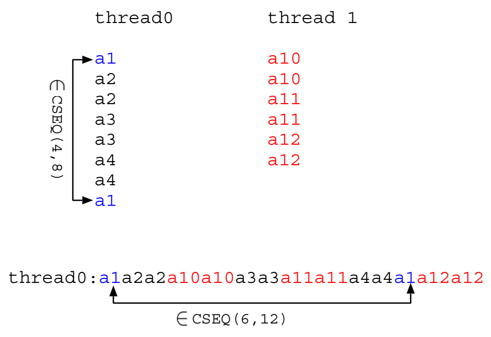

Figure 4. A cache hit of thread 0 becomes a cache miss because of references from thread 1.

The sequence at the bottom is likely to happen if we run thread 0 and thread 1 together. At this time, the second reference to $a1$ is now becoming a cache miss. Therefore, the predicted number of capacity misses on private data should be equal to the sum of the original misses and some original hits which turn into misses.

Let thread 0 and thread 1 run in parallel on two different cores. $CSEQ(d, n)$ corresponds to the cache hits of thread 0 if $d \in [1, \mathcal{C}]$. During the time thread 0 is accessing its n addresses in L2, thread 1 is also accessing the shared L2 cache. The references from thread 1 may insert an extra Δd distinct addresses into the circular sequence. When $d + \Delta d > \mathcal{C}$, thread 0's cache hit develops into a miss. For simplicity, we assume all the references inserted are different from those in the original sequence.

We use $Prob^{(pr)}_{h2m}$ to compute the interference probability for which a hit on private data becomes a miss. The modeling method for private data is an extension of the technique developed by Chandra et al. [3], which predicts the L2 cache contention for multiple processes. We apply the technique to private data profiles of shared-memory threads using:

$$Prob^{(pr)}_{h2m}(cseq^{(pr)}(d, \bar{n})) = \sum_{\hat{d} > \mathcal{C} - d} Prob(seq(\hat{d}, \bar{n})),$$

where $d \leq \mathcal{C}$ and $\bar{n}$ is the average length of sequences with d distinct addresses. Since we only consider homogeneous threads, our model scans the same trace to compute the interference probability $Prob(seq(\hat{d}, n))$. The inductive probability function used by [3] is more complex and essentially exponential. In our implementation, the computation of $\sum_{\hat{d} > \mathcal{C} - d} Prob(seq(\hat{d}, \bar{n}))$ is performed by scanning the trace file to find the frequency of sequences with length $\bar{n}$ and greater than $\mathcal{C} - d$ distinct addresses. It has a linear time complexity of $O(TraceSize)$.

This modeling process takes as input the private data circular sequence profile $CSEQ^{(pr)}(d, n)$ and works as follows:

1. Compute the total number of capacity misses when a single thread is running:

$$Misses^{(pr)}_{old} = \sum_{d > \mathcal{C}} \sum_{n > d} |CSEQ^{(pr)}(d, n)|$$

2. Compute the number of cache hits which become misses:

```
for d = 1 to $\mathcal{C}$ do
    $total\_num = \sum_{n > d} |CSEQ^{(pr)}(d, n)|$
    $\bar{n} = \frac{\sum_{n > d} (|CSEQ^{(pr)}(d, n)| \times n)}{total\_num}$
    $Prob^{(pr)}_{h2m}(d, \bar{n}) = \sum_{\hat{d} > \mathcal{C} - d} Prob(seq(\hat{d}, \bar{n}))$
    $\Delta Misses^{(pr)} += total\_num \times Prob^{(pr)}_{h2m}$
end for
```

3. Finally, compute the predicted number of capacity misses on private data:

$$Misses^{(pr)}_{new} = Misses^{(pr)}_{old} + \Delta Misses^{(pr)}$$

4.3 Modeling capacity misses on shared data

The number of capacity misses happening on the shared data is much more difficult to model than the above two types. We need to partition the shared-data circular sequence profile $CSEQ^{(sh)}(d,n)$ into two subcategories: cache hits (sequences with $d \leq \mathcal{C}$) and cache misses (sequences with $d > \mathcal{C}$). Likewise, cache hits may become misses because references from other threads stretch out the sequence length, and cache misses may become hits because other threads already loaded the data into L2. To model the two different subcategories, we adopt two different approaches, respectively.

4.3.1 Cache hits become cache misses

A thread is unable to occupy all the lines of the L2 cache when it is running concurrently with other threads. A fraction of the cache lines will contain data from the other threads. Since all threads have similar temporal behavior, we assume the effective cache size $\mathcal{C}_{eff}(t_0)$ of thread t_0 is proportional to the percentage of its footprint size to the overall footprint size:

$$\mathcal{C}_{eff}(t_0) = \frac{|footprint(t_0)|}{|\bigcup_i footprint(t_i)|} \times \mathcal{C}.$$

The number of additional cache misses $Misses_{h2m}^{(sh)}$ is computed by applying $\mathcal{C}_{eff}$ to the circular sequence profile of the concerned thread:

$$Misses_{h2m}^{(sh)} = \sum_{d=\mathcal{C}_{eff}+1}^{\mathcal{C}} \sum_{n>d} |CSEQ(d,n)|$$

Another possible approach is to use the probability model introduced in Section 4.2.

4.3.2 Cache misses become cache hits

To consider another situation where capacity misses on shared data become hits, we use the same idea as in predicting the compulsory misses. If m cache lines are accessed by n threads in common, we assume each thread will load $\frac{m}{n}$ lines. Therefore the fraction $F_{m2h}^{(sh)}$ of capacity misses that become hits is expressed as:

$$F_{m2h}^{(sh)} = 1 - \frac{1}{\text{Number of Threads}},$$

and the reduced number of capacity misses is equal to:

$$Misses_{m2h}^{(sh)} = Misses_{old}^{(sh)} \times F_{m2h}^{(sh)}.$$

By Sections 4.3.1 and 4.3.2, we can now estimate the number of capacity misses on shared data:

$$\begin{aligned} Misses_{new}^{(sh)} &= Misses_{old}^{(sh)} - Misses_{m2h}^{(sh)} + Misses_{h2m}^{(sh)} \\ &= Misses_{old}^{(sh)} \times \frac{1}{\text{Number of Threads}} \\ &\quad + Misses_{h2m}^{(sh)} \end{aligned}$$

Summing up $Misses_{new}^{(co)}$, $Misses_{new}^{(pr)}$, and $Misses_{new}^{(sh)}$ gives the predicted number of L2 cache misses.

5 Experimental results

The implementation of our model consists of a tool analyzing the L2 trace to create circular sequence profiles for each core and a library implementing the analytical model. We validate the model using three examples typical of scientific computing. All three experiments perform double-floating point operations on matrices/vectors that are stored contiguously in memory. We use the simple 1-D block data distribution to allocate tasks to two threads. The three experiments are:

- Dense matrix multiplication using three nested loops. We denote it as `dgemm`.
- Dense matrix multiplication using the tiling technique. The tile size is equal to eight. It is denoted as `blocked dgemm`.
- Sparse matrix-vector multiplication. The experiment is referred to as `spmv`.

Our experiments were conducted on an extended version of the SESC simulator. Table 1 shows the parameters of the two-core CMP architecture. A larger L2 cache results in very few capacity misses and nearly all cache misses are compulsory misses. It is relatively trivial to model such architectures. In order to model the more complicated non-compulsory misses, we choose to use a small L2 cache size.

5.1 Experimental result for dgemm

Table 2 does a comparison between the actual number of misses and the predicted number of misses for running two threads. The relative error lies in the range between 1.97% and 20.19%. For each N, there are three rows that display the actual number of L2 cache misses when we run a single thread, the actual number when we run two threads, and the prediction for running two threads, respectively.

As shown in the third row for each N, the analytical model decomposes cache misses into three components: compulsory misses, capacity misses on private data, and capacity misses on shared data. Each component adopts a different approach to model. The compulsory and shared data misses are based upon empirical parameters, and the private data misses build upon a probability model. For different applications, the total number of cache misses is dominated by one of the three components. For instance, the `dgemm` experiment has a large number of capacity misses on shared data.

5.2 Experimental result for blocked dgemm

This experiment is a tiling version of `dgemm`. It uses a block size of 8 to compute the matrix multiplication. Table 3 lists the actual number of misses for running one thread alone, the actual number for running two threads together, and the predicted number for running two threads. The relative error is between 0.1% and 4.2%.

5.3 Experimental result for spmv

Finally, we conducted experiments on sparse matrix-vector multiplications. The matrices used are `dw2048` and `qc324` which were downloaded from the Matrix Market web site. `dw2048` is a 2048×2048 sparse matrix with 10114 non-zero elements, while `qc` is a 324×324 matrix having 26730 non-zero elements. Figures 5 and 6 show their images correspondingly. Table 4 lists the performance result. To estimate the number of compulsory misses for matrix `qc324`, we observe that the two threads are working on nearly-disjoint subsets of the shared memory area, therefore we simply keep the number of compulsory misses unchanged.

Table 1. Parameters of the two-core CMP simulated architecture

Processor	Two cores, 5.0GHz out of order issue
L1(private)	ICache: LRU, 4-way, 32KB 64B line, write-through DCache: LRU, 4-way, 8KB 64B line, write-through MESI protocol
L1L2 Bus	Split transaction system bus
L2 MSHR	64
L2(shared)	Unified, LRU, 64B line 64KB, fully associative write-back

Table 2. Result for `dgemm`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each N, there are three rows. The 1st row shows the measured result for running a single thread, then the second row shows measured result for running two threads, and the third row shows our prediction.

N	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
64 single	1041	1025	16		
64 double	862	838	24		
64 predict	813	769	43	1	-5.68%
72 single	1313	1297	16		
72 double	901	870	31		
72 predict	991	973	17	1	+9.99%
80 single	1631	1601	30		
80 double	1096	1037	59		
80 predict	1233	1201	31	1	+12.50%
88 single	2076	1937	139		
88 double	6479	1158	5321		
88 predict	7787	1453	145	6189	+20.19%
96 single	56839	2305	54534		
96 double	30179	1681	28498		
96 predict	29584	1729	391	27464	-1.97%
104 single	72231	2705	69526		
104 double	35531	2037	33494		
104 predict	37710	2029	1204	34477	+6.13%
112 single	89983	3137	86846		
112 double	44428	2317	42111		
112 predict	46081	2353	607	43121	+3.72%
144 single	190445	5185	185260		
144 double	93495	3817	89678		
144 predict	97135	3889	1229	92017	+3.89%
Average Error					8.01%

6 Conclusions and future work

In this paper we present an analytical model to predict the number of L2 cache misses on a chip multi-processor quantitatively. We use the circular sequence profiling and stack processing technique to analyze an L2 cache trace. First, the trace file is scanned to generate a circular sequence profile. Next the analytical model reads in the profile and estimates the number of cache misses for running multiple threads. Since we are concentrating on a fully associative L2 cache, cache misses are decomposed into three types: compulsory misses, capacity misses on shared data, and capacity misses on private data. Each miss type is modeled by using a different method since each one's behavior is affected variously by other threads.

We have shown that the fractions of compulsory misses becoming hits and shared data capacity misses becoming hits are accurate for most of the experiments. For all the three scientific programs, the model has an average relative error less than 8.01%. In addition, the analytical model provides insight into how cache sharing and cache contention interact with each other. With this model, we are also able to predict the number of L2 cache misses for various CMP architectures given a trace. Although the model is accurate, it is not very convenient to collect the whole cache trace. We plan to further simplify the model and extend it to support heterogeneous chip multi-processors.

Table 3. Result for `blocked dgemm`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each N, there are three rows. The 1st row shows the measured result for running a single thread, then the second row shows measured result for running two threads, and the third row shows our prediction.

N	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
64 single	1047	1031	16		
64 double	805	775	30		
64 predict	827	773	53	1	+2.73%
72 single	1251	1231	20		
72 double	1952	985	977		
72 predict	1916	923	21	972	-1.84%
80 single	4537	1607	2930		
80 double	2963	1215	1748		
80 predict	3089	1205	51	1833	+4.25%
88 single	5795	1855	3940		
88 double	3374	1332	2042		
88 predict	3399	1391	71	1937	+0.74%
96 single	8161	2311	5850		
96 double	4911	1764	3147		
96 predict	4792	1733	178	2881	-2.42%
104 single	9474	2607	6867		
104 double	5319	1891	3428		
104 predict	5444	1955	108	3381	+2.35%
112 single	12690	3143	9547		
112 double	7230	2423	4807		
112 predict	7202	2357	140	4705	-0.39%
144 single	26222	5191	21031		
144 double	14543	3904	10639		
144 predict	14561	3893	299	10369	+0.12%
Average Error					1.85%

Table 4. Result for `spmv`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each sparse matrix, the 1st row shows the measured result for running a single thread, then the 2nd row shows measured result for running two threads, and the 3rd row shows our prediction.

N	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
dw single	1483	1403	80		
dw double	1483	1391	92		
dw predict	1412	1324	88	0	-4.787%
qc single	2807	2693	114		
qc double	2841	2668	173		
qc predict	2840	2693	147	0	-0.035%
Average Error					2.41%

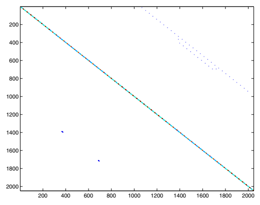

Figure 5. Sparse matrix of dw2048 ($nnz = 10,114$)

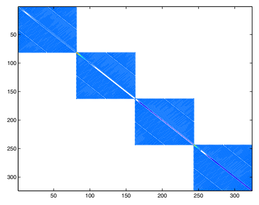

Figure 6. Sparse matrix of qc324 ($nnz = 26,730$)

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[2] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra. Matrix Market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137, 1996.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005*, pages 340–351, Feb. 2005.

[4] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.*, 64(1):108–134, 2004.

[5] A. Fedorova, M. Seltzer, and M. Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2006.

[6] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Computers*, 38(12):1612–1630, 1989.

[7] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[8] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[9] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[10] J. Renau, B. Fraguela, J. Tuck, W. Liu, and M. Prvulovic. SESC simulator, Jan. 2005. http://sesc.sourceforge.net.

[11] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[12] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 117–128, Feb. 2002.

[13] D. Thiébaut and H. Stone. Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329, 1987.