Modeling of L2 Cache Behavior for Thread-Parallel Scientific Programs on Chip Multi-Processors *

Fengguang Song, Shirley Moore, and Jack Dongarra

University of Tennessee
Computer Science Department
Knoxville, Tennessee 37996, USA
{song, shirley, dongarra}@cs.utk.edu

UT-CS-06-583

September 2006

Abstract

It is critical to provide high performance for scientific programs running on a Chip Multi-Processor (CMP). A CMP architecture often has a shared L2 cache and lower storage hierarchy. The shared L2 cache can reduce the number of cache misses if the data are commonly shared 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 a thread's performance varies when it runs together with other threads on different cores, we develop an analytical model to predict the number of misses on the shared L2 cache, especially for thread-parallel numerical codes. We assume that the parallel threads work on homogeneous tasks and share a fully associative L2 cache. Stack processing technique and circular sequences are used to analyze the L2 trace to predict the number of compulsory misses, capacity misses on shared data, and capacity misses on private data, respectively. It is the first work to predict the number of L2 misses for threads that have the nature of memory sharing. The model has been validated by three typical scientific programs: matrix multiplication, blocked matrix multiplication, and sparse matrix-vector product on a variety of matrix sizes. The average relative error lies between 2% and 12%.

*This material is based upon work supported by the National Science Foundation under grant No. 0444363.

1 Introduction

Cache performance plays an important role in software performance. With the increasing gap between memory and CPU speeds, it is more 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- 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]. Both Intel Montecito and AMD AMD64 processors can support dual-thread dual-cores [9]. Sun also shipped eight-core 32-way Niagara chips in early 2006 [7].

In these architectures, some share an on-chip L2 cache among cores and others have 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 miss only costs a few cycles. Poor L2 cache behavior can dramatically increase the amount of off-chip communication and degrade the overall performance. So 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 reuses 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 scientific applications. By analyzing the L2 cache trace which is recorded when 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 misses for threads not only from the same process, but also from different processes.

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. Terms *Shared* and *private* indicate whether the data is referenced by more than one thread (shared) or by a single thread (private). For instance, threads from different processes often have no overlapping 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 misses on private data, effective cache space and a probability method for modeling misses on shared data. Our model predicts the L2 cache behavior by finding out the cause for which a cache hit becomes a miss as well as for which a cache miss becomes a hit.

We validate the model using the cycle-accurate simulator SESC [10]. Three scientific programs have been implemented for the experiment: matrix multiplication using three nested loops, blocked matrix multiplication, and sparse matrix-vector multiplication taking as input matrices from Matrix Market [2]. The average relative error of the model is between 4% and 12%.

This paper is organized as follows: the next section introduces the related work. Section 3 outlines the concepts and techniques used by the model. Section 4 describes in detail how we model the three types of misses by different methods. In Section 5 we present 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 efficiently give miss rates for a given trace. The model is a function of a small number of factors that affect cache performance. It estimate cache performance for both a single process execution and round-robined multiple processes. Thiébaut [13] presented 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] described a technique called *stack processing* to evaluate storage hierarchies. By one-pass scanning an address trace, the frequency of stack distances can be obtained that can be used to determine the miss rate function. For the LRU algorithm, this technique works for any number of set-associativity. Suh [12] used a set of hardware counters (i.e., fully-associative coun-

ters, way-counters, or set-counters) to monitor the marginal gains in cache hits as the cache size is increased. These methods can predict the miss rate as a function of cache size, but they require that address trace be fixed. Ding [4] measured program locality by *reuse distance* and presented a two-step strategy to maximize program locality. This strategy alleviates the pressure on the insufficient memory bandwidth. Chandra [3] introduced an inductive probability model using circular sequences to predict cache interference from other threads. The probability model assumes co-scheduled threads need not share any address space. There are a lot of scientific applications that use shared-memory programming model (e.g., OpenMP programs). Our model targets at this shared-address problem and is the the first work towards solving it.

3 Methodology Overview

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 as 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 $\mathcal{C}+1$, $\mathcal{C}+2$, ..., 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

$$\mathcal{S}^{(t)} = \{s^{(t)}(1), s^{(t)}(2), \cdots, s^{(t)}(\mathcal{C})\},$$

where

$$s^{(t)}(i) = Blocks^{(t)}(\mathcal{C} = i) - Blocks^{(t)}(\mathcal{C} = i-1).$$

$s^{(t)}(i)$ is also known as "marginal gain" [12]. If a cache line x_t has been referenced before, the position of that line $\triangle$ 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 is focused only on fully-associate caches using the LRU algorithm and thus doesn't have conflict misses. It has been shown that set-associative miss ratios are related to full-associative ones and a model using Bayes rule is able to make quite accurate predictions [6]. When the number of set-associativity is large, a set-associate cache often has a miss rate comparable to a fully-associative cache.

Table 1 compares the number of extra L2 cache misses happening on a fully-associative cache with that happening on an eight-way set-associative cache for matrix multiplication with different matrix sizes. We conduct experiments on the SESC simulator. The simulated architecture has a dual-core processor, a private L1 cache on each core, and a shared L2 cache. The L2 cache is of size 64KB and has a block size of 64B. The number of extra L2 cache misses is computed by subtracting the L2 misses when two threads are running simultaneously by the L2 misses when a single thread is running. We observe that in Table 1 these two caches have similar cache behavior most of the time. Significant disagreement occurs when the matrix dimension is a multiple of the cache block size (i.e., 64 and 128), and when N=80 due to the existence of "hot sets".

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 different 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 successfully modeled

the effect of interferences from different processes. 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 (or x_n) is not observed 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 a set of circular sequences, in which each sequence is of length n and has d distinct cache lines, that is,

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

In practice we use a counter to record the number of elements in a nonempty set $CSEQ(d, n)$. Every nonempty set has a corresponding counter and the list of counters comprise a circular sequence profile.

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

Table 1. Comparison of the extra number of L2 cache misses on a fully-associative cache to that on an eight-way set-associative cache for dense matrix multiplications using three nested loops.

N	Extra Misses (fully)	Extra Misses (8-way)
64	21	849
72	30	38
80	53	544
88	4800	4363
96	165	234
104	303	372
112	243	345
128	1570	35706

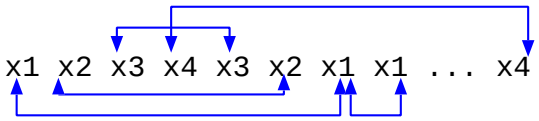

Figure 1. An example of cache block trace containing five circular sequences.

To obtain circular sequence profiles of each core, we use a simple scan process to analyze the trace. Figure 2 shows the process's corresponding C++ program, in which 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` and `cseq_private` are circular sequence counters on shared or private data for each core. This analysis process can output 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 the three components, we are able to estimate the number of misses for 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 the fully associative cache and there are no conflict misses in the model. The 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. 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"

While hardware counters are able to monitor the number of L2 cache misses, we can also use the L2 trace to estimate the number of misses as

```
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 core.

follows: The process listed in Figure 2 creates a circular sequence profile, from which we can derive the number of cache misses:

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

When 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 address location: references to the shared data among threads, and references to the thread's 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 $cseq_{shared}(4,5)$ and the second increases $cseq_{private}(4,5)$.

Different types of references are affected to different extents by other threads. When two threads are accessing the same data, the cache line previously loaded by one thread can save the other from loading it again. Therefore a compulsory miss of a thread may become a hit. Another type is that the number of capacity misses on private data will definitely increase because a hit may become a miss due to interferences from other threads. Finally, the prediction of capacity misses on shared data is more complicated. A cache miss on shared data may become a hit because the other thread has already loaded the data, meanwhile a hit may become a miss owing to other threads' interference. In the following we will describe our methods to predict these three types of misses respectively. $Misses_{new}^{(co)}$ denotes the predicted number of compulsory misses, and $Misses_{new}^{(pr)}$ denotes the predicted number of capacity misses on private data, as well as $Misses_{new}^{(sh)}$ 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 and thread 1, and a shared data access of 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's misses will probably become less 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 or actually run the two threads.

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 times the relative error is less than 15%. Figure 3 presents an example which executes 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.

When the total number of cache misses is dominated by compulsory misses, the accuracy of the

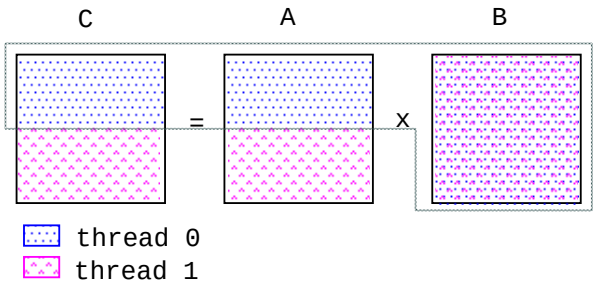

Figure 3. Two threads work on the matrix multiplication of $C = A \times B$ using block data distribution. For thread 0, half of its compulsory misses on matrix B may be saved by data loading of thread 1 (i.e., $\frac{1}{4}$ of the number of the original compulsory misses).

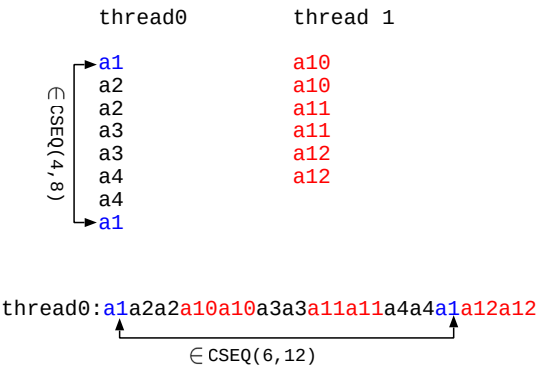

Figure 4. A cache hit of thread 0 becomes a cache miss because of references of thread 1.

assumption determines the overall precision of our model.

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

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

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

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

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

$$Misses_{new}^{(co)} = Misses_{old}^{(co)} \times F_{miss}^{(co)},$$

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

4.2 Modeling Capacity Misses on Private Data

It is easy to see that every capacity miss on private data is always a 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 capacity $\mathcal{C} = 4$. 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 $a1$ becomes a cache miss.

Let thread 0 and thread 1 run in parallel on two different cores. $CSEQ_0(d, n)$ corresponds to cache hits of thread 0 if $d \in [1, \mathcal{C}]$. We have seen that $CSEQ_0(d, n)$ is a set of circular sequences with length n and d distinct addresses. During the time thread 0 accesses the n addresses, thread 1 is also accessing the shared L2 cache. The references from thread 1 may insert extra distinct addresses Δd into the circular sequence. When $d + \Delta d > \mathcal{C}$, thread 0's hit develops into a miss. For simplicity we assume all the references inserted are different from those in the original sequence.

We use $Prob_{h2m}^{(pr)}$ to compute the probability that a hit on private data becomes a miss (extended from [3] and applied only to private-data profile):

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

where $d \leq \mathcal{C}$. Since we only consider homogeneous

threads, our model can scan the trace to compute the interference probability $Prob(seq(\hat{d}, n))$ in linear time. 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 actually performed by scanning the trace file to find the frequency of sequences with length $\bar{n}$ and greater than d distinct addresses. It has a linear time complexity of $O(TraceSize)$.

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

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

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

2. Compute the number of cache hits becoming misses:
for $d = 1$ to $\mathcal{C}$ do

$$\bar{n} = \frac{\sum_{n>d} \left(|CSEQ^{(pr)}(d, n)| \times n \right)}{\sum_{n>d} |CSEQ^{(pr)}(d, n)|}$$
$$Prob_{h2m}^{(pr)}(d, \bar{n}) = \sum_{\hat{d}>\mathcal{C}-d} Prob(seq(\hat{d}, \bar{n}))$$
$$total\_n = \sum_{n>d} |CSEQ^{(pr)}(d, n)|$$
$$\Delta Misses^{(pr)} += total\_n \times Prob_{h2m}^{(pr)}$$

end for

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

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

4.3 Modeling Capacity Misses on Shared Data

The number of capacity misses happening on the shared data between threads are more difficult to model than the above two types. We partition the shared-data circular sequence profile $CSEQ^{(sh)}(d, n)$ into two categories: hits (i.e., sequences with $d \leq \mathcal{C}$) and misses (i.e., sequences with $d > \mathcal{C}$). Similarly, hits may become misses because references from other threads stretch out the sequence length, and misses may become hits because another thread has already loaded the data into L2. We adopt two different approaches to predict these two categories.

4.3.1 Hits Become Capacity 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, the effective cache size of thread t_0 $\mathcal{C}_{eff}(t_0)$ 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 $Misses_{h2m}^{(sh)}$ which used to be hits is computed by applying $\mathcal{C}_{eff}$ to the circular sequence profile of the concerned thread:

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

4.3.2 Capacity Misses Become Hits

To consider the situation where capacity misses on shared data become hits, we use the same approach used to predict the compulsory misses described in Section 4.1. If m cache lines are commonly accessed by n threads, we assume each thread will load $\frac{m}{n}$ blocks. Therefore the fraction of capacity misses that become hits $F_{m2h}^{(sh)}$ is expressed as:

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

and the reduced number of capacity misses is equal to:

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

Based on Sections 4.3.1 and 4.3.2, we can estimate the number of capacity misses on shared

data:

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

5 Experimental Results

The implementation of our model consists of a PERL script analyzing the L2 trace to create circular sequence profiles for each core, and a C++ library realizing the analytical model. We validate the model using three examples typical of scientific computing. All three examples 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`.
- Spare matrix-vector multiplication. The experiment is referred to as `spmv`.

Our experiments are conducted on an extended version of the SESC simulator. Table 2 shows the parameters of the two-core CMP architecture.

5.1 Result for DGEMM

Table 3 does a comparison between the predicted number of misses and the actual number of misses for running two threads. The relative error lies in the range of 2% to 20% except for one case. For each N, there are three rows that display the actual number of misses when we run a single thread, the actual number when we run two threads, and the predicted number for running two threads, respectively.

For different applications, the total number of cache misses is dominated by one of the three types of misses. For instance, the DGEMM experiment has a large number of capacity misses on shared data. Please note that when N = 112, 96.5% of the original compulsory misses remain to be misses (compared between running a single thread and running two threads), instead of 75% computed by our model. The disagreement implies an imbalance problem. The immediate (direct) reason for the imbalance is that thread 0 often runs faster than thread 1 and thus loads the shared data before thread 1. Since our simulator is always deterministic, the phenomenon is likely to happen theoretically.

Among all of our experiments, this is the only exception. To further investigate it, we performed the same set of experiments on IBM Power4 and Intel Woodcrest machines and found all of them are balanced. We also improve the average relative error to 8.8% when we apply an empirical value of $F_{m2h}^{(co)}$ (e.g. based on measurement on N=72) to all the inputs. Since we are interested in an analytical approach, Table 3 does not show the improved result.

5.2 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 4 lists the actual number of misses for running one thread alone, the actual number for running two threads together, and the

Table 2. 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
L2(shared)	Unified, LRU, 64B line 64KB, fully associative write-back

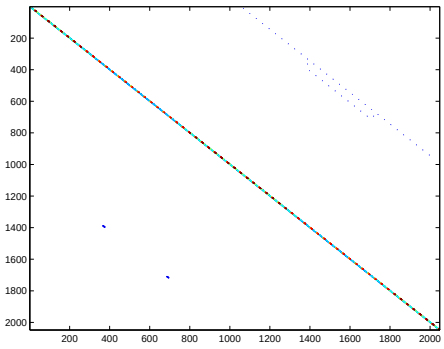

Figure 5. Sparse matrix of dw2048 ($nnz = 10,114$).

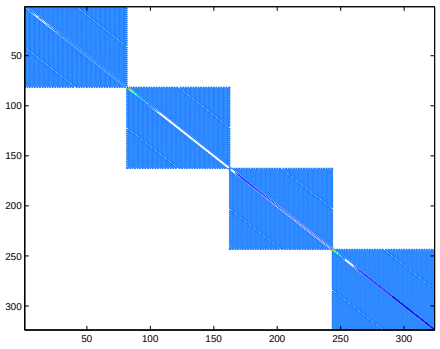

Figure 6. Sparse matrix of qc324 ($nnz = 26,730$).

predicted number for running two threads. The relative error is between 1% and 20%.

5.3 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 5 lists the performance results. 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.

6 Conclusions and Future Work

In this paper we have presented an analytical model to predict the number of L2 cache misses on a chip multi-processor quantitatively. We have used circular sequences and the stack processing technique to analyze an L2 trace. The trace file is first 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 can be decomposed into three types: compulsory misses, capacity misses on shared data, and capacity misses on private data. Each miss type is modeled using a different method since its behavior is affected variously by other threads. While the factors of $F_{m2h}^{(co)}$ and $F_{m2h}^{(sh)}$ are sometimes inaccurate, we have shown that it is accurate for most of the experiments and has a relative error less than 15%. Our ongoing work is continuing to improve its accuracy, finding the reason for the imbalance problem. In addition, we plan to integrate this quantitative model within a thread-scheduler to determine an optimal number of threads on chip multiprocessors using offline feedback-directed analysis.

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.

Table 3. 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 a single thread, then the second row measured result for two threads, and the third row shows the predicted result for the two threads.

N	Total	Compulsory	Capacity(private)	Capacity(shared)	Error
64 single	1039	1024	15		
64 double	734	709	25		
64 predict	811	768	43	0	+10.490%
72 single	1312	1296	16		
72 double	830	797	33		
72 predict	989	972	17	0	+19.156%
80 single	1632	1600	32		
80 double	1189	1125	64		
80 predict	1225	1200	25	0	+3.028%
96 single	56894	2304	54590		
96 double	27397	1541	25856		
96 predict	29248	1728	447	27073	+6.756%
104 single	72301	2704	69597		
104 double	38170	1754	36416		
104 predict	37728	2028	1223	34477	-1.157%
112 single	90078	3136	86942		
112 double	79508	3027	76481		
112 predict	46176	2352	703	43121	-41.923%
144 single	190658	5184	185474		
144 double	104360	3269	101091		
144 predict	97348	3888	1443	92017	-6.719%
Average Error					12.747%

Table 4. 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 a single thread, then the second row measured result for two threads, and the third row shows the predicted result for the two threads.

N	Total	Compulsory	Capacity(private)	Capacity(shared)	Error
64 single	1040	1024	16		
64 double	687	656	31		
64 predict	826	768	58	0	+20.2329%
80 single	4397	1600	2797		
80 double	2934	1202	1732		
80 predict	3078	1200	54	1824	+4.9080%
96 single	8161	2304	5857		
96 double	4766	1672	3094		
96 predict	4793	1728	184	2881	+0.5665%
112 single	12695	3136	9559		
112 double	7621	2380	5241		
112 predict	7209	2352	152	4705	-5.4061%
144 single	26243	5184	21059		
144 double	16794	4253	12541		
144 predict	14584	3888	327	10369	-13.1595%
Average Error					8.854%

Table 5. Result for `spmv`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each sparse matrix, there are three rows. The 1st row shows the measured result for a single thread, then the second row measured result for two threads, and the third row shows the predicted result for the two threads.

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	-0.0352%
Average Error					2.411%