A First-Order Fine-Grained Multithreaded Throughput Model

Xi E. Chen Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
{xichen, aamodt}@ece.ubc.ca

Abstract

Analytical modeling is an alternative to detailed performance simulation with the potential to shorten the development cycle and provide additional insights. This paper proposes analytical models for predicting the cache contention and throughput of heavily multithreaded architectures such as Sun Microsystems' Niagara. First, it proposes a novel probabilistic model to accurately predict the number of extra cache misses due to cache contention for significantly larger numbers of threads than possible with prior analytical cache contention models. Then it presents a Markov chain model for analytically estimating the throughput of multicore, fine-grained multithreaded architectures. The Markov model uses the number of stalled threads as the states and calculates transition probabilities based upon the rates and latencies of events stalling a thread. By modeling the overlapping of the stalls among threads and taking account of cache contention our models accurately predict system throughput obtained from a cycle-accurate performance simulator with an average error of 7.9%. We also demonstrate the application of our model to a design problem—optimizing the design of fine-grained multithreaded chip multiprocessors for application-specific workloads—yielding the same result as detailed simulations 65 times faster. Moreover, this paper shows that our models accurately predict cache contention and throughput trends across varying workloads on real hardware—a Sun Fire T1000 server.

1 Introduction

Architects typically evaluate a microprocessor design by creating a cycle-accurate simulator and then running numerous simulations to evaluate performance. As semiconductor resources continue to scale and industry develops ever larger chip multiprocessors, simulation time for cycle-accurate performance models will also grow [28], exacerbating already long development cycles.

An orthogonal approach to obtaining performance estimates is analytical modeling [27]. An analytical model employs mathematical formulas that approximate the performance of the processor being designed based upon program characteristics and microarchitectural parameters. While workload sampling [21], parallel simulation [19], and FPGA acceleration [5, 25] can reduce simulation time, these approaches all require the creation of detailed performance models before any feedback on a proposed microarchitecture change is available to the design team.

Several analytical models have been proposed for single-threaded processors [4, 11, 14, 16, 17, 18], but very little work has focused on how to analytically model the *throughput* of multithreaded processors. In this paper we propose analytical models for predicting cache contention and overall throughput for fine-grained multithreaded architectures similar to Sun Microsystems' Niagara [13]. While recent research has explored analytical modeling of cache contention between two threads on a dual core system [3], we find that this approach becomes inaccurate for systems with a large number of threads, particularly once the number of threads sharing a cache approaches or exceeds the associativity of the cache (see Figure 2).

This paper makes the following four contributions:

- It proposes a novel probabilistic cache contention model to accurately predict the number of extra cache misses due to cache contention for a large number of threads sharing a cache.
- It presents four analytical models of varying accuracy and complexity for estimating the throughput of a multicore, fine-grained multithreaded single-issue processor (similar to Sun's Niagara). The most accurate model utilizes a Markov chain (example in Figure 5) to leverage the cache contention model we propose.
- It applies the combined cache contention and Markov chain throughput model to optimize a multicore fine-grained multithreaded processor for two different application-specific workloads yielding the same design points as detailed simulation.

• Finally, it validates the models against real har ware—a Sun Fire T1000 server.

In addition to being useful for throughput modeling, c novel cache contention model may also significantly e tend the variety of hardware systems for which an operati system can apply an analytical cache contention model determine co-scheduled threads to reduce cache contenti (e.g., as proposed by [3]).

The rest of this paper is organized as follows: Sectior reviews a prior model for predicting cache contention b tween two threads and then describes our novel cache contention model for a large number of threads. Section 3 proposes several analytical models for estimating the throughput of fine-grained multithreaded architectures. Section 4 describes our evaluation methodology. Section 5 analyzes our experimental results including an application of our analytical models to architecture optimization. Section 6 reviews related work. Finally, Section 7 concludes.

2 Modeling Cache Contention

In this section, we first summarize a prior cache contention model [3], then explain and evaluate its limitations. Next, we propose two novel metrics for quantifying aspects of temporal locality necessary to overcome these limitations. Finally, we propose a novel cache contention model using these locality metrics to accurately model cache contention for a large number of threads.

2.1 A Prior Cache Contention Model

Chandra et al. [3] propose a probabilistic model for predicting the number of extra cache misses due to cache contention between two threads on a chip multiprocessor architecture. The model uses information from a circular sequence profile obtained for each thread running alone. In their work, a circular sequence is defined as a sequence of accesses to the same cache set from a thread such that the first and the last access are to the same cache block, which is different from all the blocks touched by intermediate accesses. Figure 1 illustrates the notion of a circular sequence. In Figure 1 the blocks A, B, C, D, and E map to the same set in a four-way LRU cache. A circular sequence is denoted as $\mathtt{cseq(d,n)}$, where d is the number of distinct blocks in the circular sequence, n is the total number of blocks in the circular sequence, and d is at least one less than n (as a result of the definition of $\mathtt{cseq(d,n)}$).

In an x-way associative cache with LRU replacement the last access in a circular sequence $\mathtt{cseq(d,n)}$ is a hit if and only if $\mathtt{d} \leq \mathtt{x}$. Thus, by performing circular sequence profiling it is possible to determine whether a load or store misses in the cache. In Figure 1 we assume a four-way LRU cache. Here, the last access (to block A) results in a cache miss since it corresponds to $\mathtt{cseq(5,6)}$ and 5 is greater than the cache associativity. For all the other circular sequences in Figure 1, the last access corresponds to a cache hit.

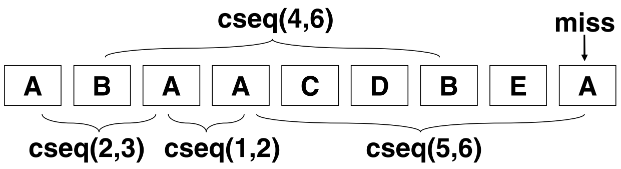


Figure 1. An example of circular sequences in a set of a four-way associative LRU cache

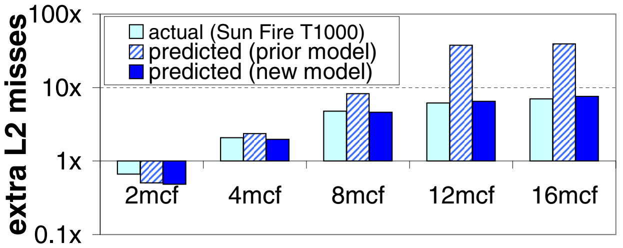


Figure 2. Extra L2 cache misses due to cache contention when multiple copies of *mcf* are running in parallel (L2 config. in Table 2)

To model the number of extra cache misses due to cache contention, a circular sequence profile is performed to obtain the distribution of $\mathtt{cseq(d,n)}$ for a given cache configuration. Using this information, the average of n weighted by the frequency of occurrence of $\mathtt{cseq(d,n)}$, denoted $\overline{\mathtt{n}}$, is computed for each d between 1 and x (where x is the cache associativity). After obtaining $\overline{\mathtt{n}}$ for each d, the prior model uses the *access frequency* per set (average number of accesses per cycle to the cache set, including both hits and misses) to approximate the average time interval (in cycles) between the first and the last access of $\mathtt{cseq(d,*)}$. Here $\mathtt{cseq(d,*)}$ represents all circular sequences containing d distinct tags. Next, the estimated time interval is converted to the number of cache accesses to that set from the second thread using the the second thread's access frequency. Then, an inductive model is used to estimate the probability that a specified number of distinct blocks are accessed in the set from the second thread. Finally, the probability that a cache hit from the first thread becomes a miss due to interference from the second thread is approximated, and the number of extra cache misses of the first thread is calculated.

The prior model works well when there are only two threads sharing a cache with a high associativity. As the number of threads sharing the cache increases, the model becomes less accurate. Figure 2 compares, for multiple copies of *mcf* running in parallel, the predicted result both from an extension to the prior model [3] (labeled "predicted (prior model)") and from our new model described in Section 2.3 (labeled "predicted (new model)") to the result measured on real hardware–a Niagara T1 based Sun Fire T1000 system. The predicted result is the ratio of the number of *extra* L2 misses per *mcf* due to cache contention to the num-

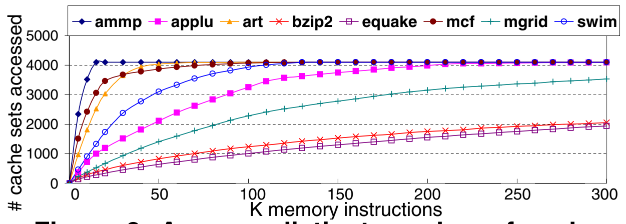


Figure 3. Average distinct number of cache sets accessed during a given number of consecutive memory instructions for a twelve-way, 64B line, 4096-set LRU L2 cache

ber of misses when *mcf* runs alone. Note we have extended the *Prob* approach presented for two threads in Chandra et al. [3] by considering the interleaving of sequences from more than one interfering thread[1]. In Figure 2, we observe that the error of the prior model increases significantly as the number of threads sharing the cache increases. For example, consider the case where twelve threads share a twelve-way L2 cache (i.e., 12mcf in Figure 2). For any given circular sequence of a thread, the number of accesses from each co-scheduled thread occurring between the first and the last access in the circular sequence is predicted to be *at least* one by the prior model. Therefore, only a small fraction of the last accesses in $\texttt{cseq}(1,*)$ circular sequences are modeled as hits (when each of the eleven co-scheduled threads inserts only one access in a $\texttt{cseq}(1,*)$). In this example, the actual number of L2 misses increases by a factor of 6.2× and our new model presented in Section 2.3 predicts an increase of 6.5× (4.8% error). However, the prior model predicts an increase of 37.6× since it predicts 95.7% of the accesses that hit in the cache when *mcf* runs alone will turn into cache misses (resulting in a 507.9% error).

2.2 New Locality Metrics for Modeling Cache Contention

While circular sequence profiling captures many aspects of temporal locality, what is missing in the prior model described in Section 2.1 is the fact that the cache footprint of a thread during a small interval of time is likely to be limited to a small number of cache sets rather than all of them. If the time interval between the first and the last access in a circular sequence is short, cache accesses from other co-scheduled threads are not likely to be sent to the same cache set containing the accesses from the circular sequence. Therefore, even in the presence of many co-scheduled threads the last access in a circular sequence may not turn into a miss.

To quantify the above behavior we introduce the parameter $\texttt{S}(\texttt{x})$, which is the average number of sets accessed by a thread during $\texttt{x}$ consecutive loads or stores. This can be obtained by off-line profiling a memory access trace for a specific cache configuration (or potentially in hardware—e.g., by adding a single bit per cache set along with a single counter and a table—though detailed explorations of such implementations are beyond the scope of this work).

Figure 3 shows $\texttt{S}(\texttt{x})$ of different benchmarks for the L2 cache we model in Section 5. From the figure we observe that as the number of consecutive memory instructions being analyzed increases, the number of distinct cache sets accessed increases until saturating at the total number of cache sets (provided the data set of an application is large enough to use all sets in the cache). A thread with higher $\texttt{S}(\texttt{x})$ for a given $\texttt{x}$ is more likely to access sets currently used by other threads.

Besides the number of cache sets accessed, a thread's cache footprint also depends on the number of distinct blocks accessed in those sets. For example, it is possible that there are two threads with the same value of $\texttt{S}(\texttt{x})$ for a fixed $\texttt{x}$ (number of memory instructions). However, for each set being accessed, there are ten distinct blocks from the first thread and only two from the second thread. Then, although the two threads access the same number of sets, the first thread's cache footprint over this interval would be five times larger than the second thread.

To take the number of distinct blocks in a set accessed by a thread into consideration, we propose using a probability vector, $\texttt{b}(\texttt{i},\texttt{x})$, to estimate the probability that there are $\texttt{i}$ distinct accesses to a cache set during $\texttt{x}$ consecutive memory instructions, given that the cache set is accessed. Similar to $\texttt{S}(\texttt{x})$, $\texttt{b}(\texttt{i},\texttt{x})$ can be obtained via off-line profiling. Figure 4 illustrates $\texttt{b}(\texttt{i},\texttt{x})$ with different values of $\texttt{x}$ for *mcf*. For each $\texttt{x}$, the probability that there are one to eleven (i.e., associativity-1) distinct accesses to a cache set is shown in the first eleven bars, while the rightmost bar of each figure is the probability of having more than or equal to twelve (i.e., associativity) distinct accesses.

From Figure 4, we observe that when the measurement interval (i.e., $\texttt{x}$) is small, the average number of distinct accesses in each cache set being accessed is likely to be small. As $\texttt{x}$ increases, the mode in each $\texttt{b}(\texttt{i},\texttt{x})$ distribution tends to move toward higher values of $\texttt{i}$. When $\texttt{x}$ is sufficiently large, the number of distinct accesses is typically greater than or equal to the associativity of the cache. Our use of $\texttt{b}(\texttt{i},\texttt{x})$ essentially replaces the use of the inductive probability model [3]. In the next section, we utilize $\texttt{S}(\texttt{x})$, $\texttt{b}(\texttt{i},\texttt{x})$, along with circular sequence profiling to more accurately quantify the additional number of cache misses due to contention among threads when the number of co-scheduled threads approaches or exceeds the associativity of the cache

[1]For "prior model" in Figure 2 we use $P_{miss}(cseq_{X_1}(d_{X_1}, n_{X_1})) = 1 - \sum_{d_{X_2}+\ldots+d_{X_N} \le A-d_{X_1}} \prod_{i=2\ldots N} P(seq(d_{X_i}, E(n_{X_i})))$ to extend Equation 3 for the *Prob* model in Chandra et al. [3] for more than two threads. Here X_i, $i \neq 1$, represents each interfering thread, A is the cache associativity, and N is the total number of threads. This equation follows directly after extending Corollary 3 in [3], such that the last access from X_1 in $cseq_{X_1}(d_{X_1}, n_{X_1})$ results in a miss if $\sum_{i=1}^{N} d_{X_i} > A$ (or a hit otherwise).

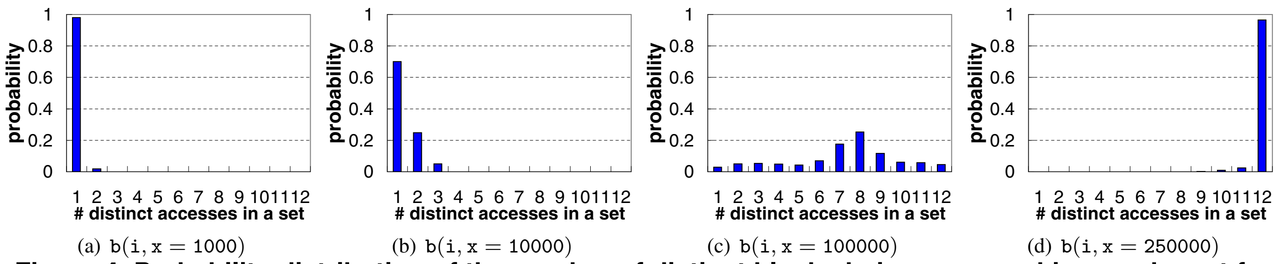


Figure 4. Probability distribution of the number of distinct blocks being accessed in a cache set for *mcf* on a 12-way, 64B line, 3 MB cache. Horizontal axis is value of i, and x is the measurement interval counted in consecutive memory instructions

being shared.

2.3 Accurately Modeling Cache Contention with Many Threads

For concreteness, we will describe how to model the number of extra cache misses of a thread (T1) due to sharing a four-way set associative L1 data cache with three other threads (T2, T3, and T4). We also generalize the details for different numbers of threads and cache associativities.

We define the *distance* between two accesses from a single thread as the total number of intermediate memory accesses from the thread (i.e., independent of which cache sets are accessed). As mentioned in Section 2.2, for a circular sequence $\mathtt{cseq(d, n)}$, the higher the distance between its first and last access, the more likely the last access will become a miss due to cache contention. Therefore, hereafter we describe each circular sequence with the distance between its first and last access as $\mathtt{cseq(d, n, r)}$, where $\mathtt{r}$ represents the distance. We use $\mathtt{cseq(d, *, r)}$ for the set of circular sequences with $\mathtt{d}$ distinct tags, for all $\mathtt{n}$ such that $\mathtt{d} < \mathtt{n} \leq \mathtt{r}$.

To analytically model cache contention, we carry out the following steps twice: Once with rough CPI estimates using the number of cache misses in the absence of contention from a cache simulator for a single thread; and, again after feeding back our first estimate of the number of extra cache misses to obtain a more accurate CPI estimate. For the purpose of estimating the number of extra misses due to cache contention, we estimate per-thread CPI by simply multiplying the number of misses and miss penalty [9] as our cache contention model only requires the relative rate of progress among threads. For each value of $\mathtt{d}$ between one and four (i.e., the associativity of the L1 cache), we carry out the following six steps:

- **Step 1**: Apply circular sequence profiling and classify circular sequences $\mathtt{cseq(d, n, r)}$ into different groups based upon $\mathtt{r}$. Each group $\mathtt{g}$ corresponds to a range of $\mathtt{r}$. Then, for a group $\mathtt{g}$, calculate $\overline{\mathtt{dist}}_{\mathtt{(d,g)}}$, the weighted $\mathtt{r}$ (distance between the first and last access in a circular sequence) for all circular sequences $\mathtt{cseq(d, n, r)}$ belonging to the group as follows:

$$\overline{\mathtt{dist}}_{\mathtt{(d,g)}} = \frac{\sum_{\mathtt{r}=\underline{\mathtt{r}}_{\mathtt{g}}}^{\overline{\mathtt{r}}_{\mathtt{g}}} [\mathtt{r} \times \mathtt{num(cseq(d, *, r))}]}{\sum_{\mathtt{r}=\underline{\mathtt{r}}_{\mathtt{g}}}^{\overline{\mathtt{r}}_{\mathtt{g}}} \mathtt{num(cseq(d, *, r))}}$$

  where $\underline{\mathtt{r}}_{\mathtt{g}}$ and $\overline{\mathtt{r}}_{\mathtt{g}}$ represents the lower and the upper threshold of the group $\mathtt{g}$, respectively, and $\mathtt{num(cseq(d, *, r))}$ denotes the number of occurrences of the circular sequence $\mathtt{cseq(d, *, r)}$ observed during profiling. In our study, we classify a circular sequence into one of twelve groups. The first group corresponds to all the circular sequences with $1 < \mathtt{r} < 2^5$, the next ten groups with $2^{\mathtt{i}} \leq \mathtt{r} < 2^{\mathtt{i}+1}, \mathtt{i} \in 5, 6, ..., 14$, and the last group corresponds to all the circular sequences with $\mathtt{r} \geq 2^{15}$. Using more than one group improves accuracy, but we have not explored this space further.

- **Step 2**: For each $\overline{\mathtt{dist}}_{\mathtt{(d,g)}}$, calculate $\mathtt{n}_{\mathtt{T2(d,g)}}$, $\mathtt{n}_{\mathtt{T3(d,g)}}$, $\mathtt{n}_{\mathtt{T4(d,g)}}$, the corresponding numbers of accesses for $\mathtt{T}_2$, $\mathtt{T}_3$, and $\mathtt{T}_4$, respectively, based upon the access frequency of those threads:

$$\mathtt{n}_{\mathtt{Ti(d,g)}} = \overline{\mathtt{dist}}_{\mathtt{(d,g)}} \times \frac{\mathtt{access\_freq(T_i)}}{\mathtt{access\_freq(T_1)}} \quad \mathtt{for\ i = 2,\ 3,\ 4}$$

  where $\mathtt{access\_freq(T_i)}$ is the number of memory instructions per cycle for thread $\mathtt{T_i}$ estimated from a simple single thread, sum of CPI components model using the (current best) estimated number of cache misses.

- **Step 3**: Using $\mathtt{n}_{\mathtt{Ti(d,g)}}$ from Step 2, find $\mathtt{S}_{\mathtt{Ti(d,g)}}$, the average number of cache sets accessed during $\mathtt{n}_{\mathtt{Ti(d,g)}}$ consecutive memory access instructions, by using $\mathtt{S(x)}$ described in Section 2.2:

$$\mathtt{S}_{\mathtt{Ti(d,g)}} = \mathtt{S(n_{Ti(d,g)})} \quad \mathtt{for\ i = 2,\ 3,\ 4}$$

- **Step 4**: Using $\mathtt{n}_{\mathtt{Ti(d,g)}}$ from Step 2 and $\mathtt{b(k, n)}$ (defined in Section 2.2), find $\mathtt{d}_{\mathtt{Ti(d,g)}}\mathtt{(k)}$—the distribution of unique cache blocks accessed in a set by thread $\mathtt{T_i}$—as follows:

$$\mathtt{d}_{\mathtt{Ti(d,g)}}\mathtt{(k)} = \mathtt{b(k, n_{Ti(d,g)})}$$

$$\mathtt{for\ i = 2,\ 3,\ 4\ (\#threads);\ k = 1,\ 2,\ 3,\ 4\ (assoc.)}$$

- **Step 5**: For each group of circular sequences, calculate $\mathtt{prob_H(d, g)}$, the probability that the last access of a circular sequence $\mathtt{cseq(d, n, r)}$ in group $\mathtt{g}$ is *not* turned into a miss due to cache contention. To model the probability that another thread ($\mathtt{T_i}$) accesses the cache set containing the circular sequence, we divide $\mathtt{S}_{\mathtt{Ti(d,g)}}$, obtained from Step 3, by the total number of cache sets (which we will represent with $\mathtt{S}$) to obtain $\frac{\mathtt{S}_{\mathtt{Ti(d,g)}}}{\mathtt{S}}$. The intuition here is that the chance any given set (i.e., the set containing the circular sequence) is accessed by a thread during a fixed time interval is proportional to the number of distinct sets accessed by the thread during that amount of time. We compute the desired probability as the sum of four parts (since there are four threads per core): **(i)** the probability that neither $\mathtt{T}_2$, $\mathtt{T}_3$, or $\mathtt{T}_4$ accesses the set containing the circular sequence, **(ii)** the probability that only one of $\mathtt{T}_2$, $\mathtt{T}_3$, or $\mathtt{T}_4$ accesses the set containing the circular sequence and the number of distinct accesses to the set from the thread is less than or equal to the difference between the cache's associativity $\mathtt{A}$ and $\mathtt{d}$ (i.e., the maximum number of distinct accesses from other threads without turning the last access of the circular sequence into a miss), **(iii)** the probability that two of $\mathtt{T}_2$, $\mathtt{T}_3$, or $\mathtt{T}_4$ access the set and the sum of the number of distinct accesses to the set from the two threads is less than or equal to the difference $\mathtt{A} - \mathtt{d}$, and **(iv)** the probability that $\mathtt{T}_2$, $\mathtt{T}_3$, and $\mathtt{T}_4$ all access the set and the sum of the number of distinct accesses to the set from all three threads is less than or equal to the difference $\mathtt{A} - \mathtt{d}$.

The general formula for computing $\mathtt{prob}_\mathtt{H}(\mathtt{d}, \mathtt{g})$ for arbitrary number of threads $\mathtt{N}$, and associativity $\mathtt{A}$ is[2]:

$$\prod_{i=2}^{i=N}(1 - \frac{S_{T_i(d,g)}}{S}) + \sum_{j=1}^{N-1}\Bigg\{ \sum_{\substack{2\le i_1,i_2,\ldots,i_{N-1}\le N \\ i_1\ne i_2\ldots\ne i_{N-1} \\ \forall\alpha_{i_n}\in 1..\mathtt{A},\, \sum_{n=j}^{N-1}\alpha_{i_n}\le \mathtt{A}-\mathtt{d}}} \left[\prod_{n=1}^{j-1}(1 - \frac{S_{T_{i_n}(d,g)}}{S}) \prod_{n=j}^{N-1} \frac{S_{T_{i_n}(d,g)}}{S} d_{T_{i_n}(d,g)}(\alpha_{i_n})\right]\Bigg\}$$

- **Step 6**: Finally, the total number of extra misses is calculated as:

  $$\sum_{\mathtt{d}=1}^{4}\sum_{\mathtt{g}=1}^{12}[(1 - \mathtt{prob}_\mathtt{H}(\mathtt{d},\mathtt{g})) \times \sum_{\mathtt{r}=\underline{\mathtt{r}}_\mathtt{g}}^{\overline{\mathtt{r}}_\mathtt{g}} \mathtt{num}(\mathtt{cseq}(\mathtt{d},*,\mathtt{r}))]$$

  Here $\mathtt{4}$ is the cache associativity and $\mathtt{12}$ is the number of groups that we use to classify a circular sequence.

For our study, we implement the above calculations in Matlab and the running time (on an Intel Core 2 Duo desktop system) is on the order of 0.3 seconds.

[2]For example, consider $\mathtt{d} = 2$ for $\mathtt{N} = 4$ and $\mathtt{A} = 4$. To compute $\mathtt{prob}_\mathtt{H}(2, \mathtt{g})$ the first component is: $(1 - \frac{S_{T_2(2,g)}}{S})(1 - \frac{S_{T_3(2,g)}}{S})(1 - \frac{S_{T_4(2,g)}}{S})$ the second component is: $(\frac{S_{T_2(2,g)}}{S} d_{T_2(2,g)}(1))(1 - \frac{S_{T_3(2,g)}}{S})(1 - \frac{S_{T_4(2,g)}}{S}) + (1 - \frac{S_{T_2(2,g)}}{S})(\frac{S_{T_3(2,g)}}{S} d_{T_3(2,g)}(1))(1 - \frac{S_{T_4(2,g)}}{S}) + (1 - \frac{S_{T_2(2,g)}}{S})(1 - \frac{S_{T_3(2,g)}}{S})(\frac{S_{T_4(2,g)}}{S} d_{T_4(2,g)}(1)) + (\frac{S_{T_2(2,g)}}{S} d_{T_2(2,g)}(2))(1 - \frac{S_{T_3(2,g)}}{S})(1 - \frac{S_{T_4(2,g)}}{S}) + (1 - \frac{S_{T_2(2,g)}}{S})(\frac{S_{T_3(2,g)}}{S} d_{T_3(2,g)}(2))(1 - \frac{S_{T_4(2,g)}}{S}) + (1 - \frac{S_{T_2(2,g)}}{S})(1 - \frac{S_{T_3(2,g)}}{S})(\frac{S_{T_4(2,g)}}{S} d_{T_4(2,g)}(2))$ and the third component is $(\frac{S_{T_2(2,g)}}{S} d_{T_2(2,g)}(1))(\frac{S_{T_3(2,g)}}{S} d_{T_3(2,g)}(1))(1 - \frac{S_{T_4(2,g)}}{S}) + (\frac{S_{T_2(2,g)}}{S} d_{T_2(2,g)}(1))(1 - \frac{S_{T_3(2,g)}}{S})(\frac{S_{T_4(2,g)}}{S} d_{T_4(2,g)}(1)) + (1 - \frac{S_{T_2(2,g)}}{S})(\frac{S_{T_3(2,g)}}{S} d_{T_3(2,g)}(1))(\frac{S_{T_4(2,g)}}{S} d_{T_4(2,g)}(1))$, while the last component is zero since a $\mathtt{cseq}(2, \mathtt{n}, \mathtt{r})$ can at most tolerate two accesses with a four-way associative cache.

3 Modeling Throughput

In this section, we present several analytical models for predicting the throughput of fine-grained multithreaded architectures. The models proposed in this paper apply directly to multiprogrammed workloads in which threads do not communicate with each other (i.e., incur synchronization overheads). To model performance of parallel workloads in which threads communicate, the models proposed here could be employed as the *lower-level system model* of a two-level hierarchical parallel performance model such as the one proposed by Adve and Vernon [1]. In such models, the lower-level system model is invoked to predict the throughput of the system up until the next synchronization point. Moreover, if an OS performs unrelated processing on some thread contexts, one could treat the OS as a thread with a different memory access pattern while any synchronization among threads via the OS could potentially be modeled by incorporating our model with the two-level hierarchical model mentioned above.

3.1 Sum of Cycles Model

One approach to predicting overall throughput is to simply assume no overlapping of execution occurs on a core. While this assumption is pessimistic for memory-bound applications, it does match the case where an application is entirely compute-bound. We evaluate this model using: $\mathtt{throughput}_\mathtt{C} = \frac{\sum_{\mathtt{i}=1}^{\mathtt{N_C}} \mathtt{num\_inst}_\mathtt{Ti}}{\sum_{\mathtt{i}=1}^{\mathtt{N_C}} \mathtt{cycle}_\mathtt{Ti}}$, where $\mathtt{throughput}_\mathtt{C}$ is the throughput on core $\mathtt{C}$, $\mathtt{num\_inst}_\mathtt{Ti}$ is the number of instructions executed by thread $\mathtt{Ti}$, $\mathtt{cycle}_\mathtt{Ti}$ is the number of cycles taken by the thread to run those instructions when the thread runs alone and $\mathtt{N_C}$ is the total number of threads running on core $\mathtt{C}$. In Section 5 we will show that this model always underestimates the real throughput of a fine-grained multithreaded core on our workloads (by 58% on average).

3.2 Sum of IPCs Model

The model in Section 3.1 always underestimates throughput since it ignores the fact that stalls from different threads on a core can overlap. To overcome this drawback, a simple approach is to sum up each thread's isolated *instructions per cycle* (IPC)—the IPC when the thread runs alone. This works well when the execute stage of a fine grained multithreaded pipeline is lightly utilized by a single thread. Such behavior is common for memory-bound applications. However, we find that this model will always overestimate the real throughput (by 55% on average for our workloads as shown in Section 5).

3.3 Bernoulli Model

Both models above oversimplify pipeline resource sharing in fine-grained multithreaded architectures. To more

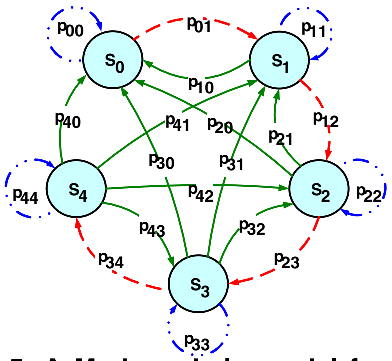


Figure 5. A Markov chain model for a four-thread fine-grained multithreaded architecture

accurately model overlapping of stalls from threads running on a core, we assume that every cycle each thread performs a "coin toss" (or Bernoulli trial) to determine whether it has an instruction ready to execute. Ignoring the effects of resource contention between threads, the probability that a thread $\mathtt{T_i}$ has an instruction ready to execute can be estimated as $\mathtt{IPC_{Ti}}$ (the thread's isolated IPC) since this is a value between zero and one for a single-issue pipeline. Then $1 - \mathtt{IPC_{Ti}}$ can be used to estimate the probability that $\mathtt{T_i}$ cannot issue an instruction on any given cycle and $\prod_{i=1}^{N_C}(1 - \mathtt{IPC_{Ti}})$ becomes an estimate for the probability that *no* thread has an instruction ready to issue on a given cycle, where $\mathtt{N_C}$ is the number of threads running on a core and $\mathtt{IPC_{Ti}}$ represents the isolated IPC of thread Ti. Thus, the throughput of fine-grained multithreaded core C can be estimated as the product of the peak issue width (1.0) and the probability that *at least* one thread can issue an instruction in a cycle: $\mathtt{throughput_C} = 1.0 \times (1 - \prod_{i=1}^{N_C}(1 - \mathtt{IPC_{Ti}}))$. Concurrent with our work, Govindaraju et al. proposed a customized analytical model [8] similar to our Bernoulli model. In Section 5 we will show that our Bernoulli model reduces error to an overestimate of real throughput of 23% on average.

3.4 A Markov Chain Model

The techniques proposed from Section 3.1 to 3.3 do not model contention in the memory system or the time dependence of long latency stall events. To account for these factors, we use a Markov chain to model the partial hiding of stalls of a thread by executions from other threads. We will show in Section 5 that, when combined with our novel cache contention model proposed in Section 2.3, the Markov chain model proposed in this section reduces the average error of modeled throughput to 7.9% compared against detailed simulation.

Figure 5 illustrates an example of the Markov chain we use to model throughput for the case of a four-thread, single-issue, in-order, fine-grained multithreaded pipeline with a memory system that supports at most one outstanding long latency cache miss per thread. We note that the assumption of one outstanding cache miss per thread does match the designs of Sun's Niagara T1 and T2 processors [15]. Experiments with our detailed simulator found that supporting additional outstanding cache misses per thread improved performance by 4.6% on average for the configuration and workloads described in Table 2 and Table 3. Extending our models for multiple outstanding cache misses per thread is beyond the scope of this study and is left for future work.

The Markov chain illustrated in Figure 5 has five states labeled from $\mathtt{S_0}$ to $\mathtt{S_4}$, where state $\mathtt{S_n}$ corresponds to the case that n threads are currently suspended from issuing new instructions because of a prior long latency event such as a cache miss. In state $\mathtt{S_0}$ no thread is suspended (i.e., all threads are ready to issue instructions) whereas $\mathtt{S_4}$ means that all four threads are suspended (i.e., no thread is currently able to issue instructions). In any cycle, if there is at least one thread ready to issue then we assume one thread will issue an instruction. Thus, the throughput of an N-thread fine-grained multithreaded core can be modeled as $1 - \mathtt{prob}(\mathtt{S_N})$, where the term $\mathtt{prob}(\mathtt{S_N})$ represents the probability of being at $\mathtt{S_N}$ (no threads ready—$\mathtt{S_4}$ in Figure 5).

From each state, only certain transitions are possible since only one thread can execute in a given cycle. The allowed transitions among states for a four-thread fine-grained multithreaded core are shown in Figure 5. There are three types of transitions: upstream transitions (illustrated by broken lines, e.g., $\mathtt{p_{01}}$), downstream transitions (illustrated by solid lines, e.g., $\mathtt{p_{10}}$), and staying transitions (illustrated by broken lines separated by dots, e.g., $\mathtt{p_{00}}$). An *upstream transition* corresponds to $\mathtt{S_x} \rightarrow \mathtt{S_y}$, where $\mathtt{x} < \mathtt{y}$; a *downstream transition* corresponds to $\mathtt{S_x} \rightarrow \mathtt{S_y}$, where $\mathtt{x} > \mathtt{y}$; and a *staying transition* does not change states[3]. At most one thread can become newly suspended on a cycle, thus preventing transitions such as $\mathtt{S_0} \rightarrow \mathtt{S_2}$, while multiple threads can be reactivated in the same cycle (e.g., one thread finishes a long latency floating point operation and another thread gets its required data from memory).

To determine the state transition probabilities we first consider homogeneous workloads in Section 3.4.1 and then heterogeneous workloads in Section 3.4.2.

3.4.1 Homogeneous Workloads

In this section we describe how to obtain the state transition probabilities $\mathtt{p_{ij}}$ assuming that all threads have the same program characteristics so that the probability of being suspended is the same for each thread. To simplify the discussion we also assume only one type of event can stall

[3]Although the state is not changed, some events might occur. For example, it is possible that the state remains unchanged while one suspended thread is reactivated and another thread becomes newly suspended.

Table 1. Transition probability definitions for an N-thread fine-grained multithreaded architecture

Upstream $(j > i)$	Value	(#susp., #act.)
$j = i + 1$	$p_{ij} = p(\frac{M-1}{M})^i$	(1,0)

Downstream $(j < i)$	Value	(#susp., #act.)
$i \neq N,\ \ j \neq 0$	$p_{ij} = (1-p)\binom{i}{i-j}(\frac{1}{M})^{i-j}(\frac{M-1}{M})^j + p\binom{i}{i-j+1}(\frac{1}{M})^{i-j+1}(\frac{M-1}{M})^{j-1}$	(0,i-j) or (1,i-j+1)
$i \neq N,\ \ j = 0$	$p_{ij} = (1-p)\binom{i}{i-j}(\frac{1}{M})^{i-j}(\frac{M-1}{M})^j$	(0,i-j)
$i = N$	$p_{ij} = \binom{i}{i-j}(\frac{1}{M})^{i-j}(\frac{M-1}{M})^j$	(0,i-j)

Staying $(j = i)$	Value	(#susp., #act.)
$i \neq N,\ \ i \neq 0$	$p_{ii} = (1-p)(\frac{M-1}{M})^i + p\binom{i}{1}(\frac{1}{M})(\frac{M-1}{M})^{i-1}$	(0,0) or (1,1)
$i = 0$	$p_{ii} = 1 - p$	(0,0)
$i = N$	$p_{ii} = (\frac{M-1}{M})^i$	(0,0)

a thread. We will relax both assumptions in Section 3.4.2.

To begin, we define two parameters: p and M. The term p represents the probability of a thread being suspended. For homogeneous workloads p is the same for all threads and can be calculated as the fraction of instructions causing a thread to be suspended. The term M represents the latency (in cycles) of the event causing a thread to be suspended. We model the probability of reactivating a suspended thread on any given cycle as $\frac{1}{M}$ since, in the next M cycles, the suspended thread can only be reactivated in the last cycle. Thus, the probability that a suspended thread remains suspended on a given cycle is $1 - \frac{1}{M} = \frac{M-1}{M}$.

Table 1 summarizes the transition probabilities and lists (in the third column) how many threads become newly suspended and/or activated for each possible transition. For example, consider the upstream transition p_{23} (i.e., the probability of $S_2 \rightarrow S_3$) is modeled as $p(\frac{M-1}{M})^2$, where p corresponds to the probability that one thread becomes newly suspended, and $(\frac{M-1}{M})^2$ corresponds to the probability that both the threads that have already been suspended remain suspended. The term $(1,0)$ in the third column indicates that one thread becomes newly suspended and no threads are newly activated for this transition.

Some state transitions can be achieved in more than one way. For instance, the downstream transition p_{32} (i.e., the probability of $S_3 \rightarrow S_2$), is modeled as $(1-p)\binom{3}{1}(\frac{1}{M})(\frac{M-1}{M})^2 + p\binom{3}{2}(\frac{1}{M})^2(\frac{M-1}{M})$. In the term before the plus sign, $(1-p)$ is the probability that the currently executing thread does not become suspended and $\binom{3}{1}(\frac{1}{M})(\frac{M-1}{M})^2$ is the probability that, in the three threads that have already been suspended, one thread is reactivated this cycle (probability $\frac{1}{M}$) and the other two remain suspended (probability $(\frac{M-1}{M})^2$). Note that $\binom{3}{1}$ is the number of ways to choose the one thread that is activated out of three threads. In the term after the plus sign, p is the probability that the current executing thread becomes newly suspended and $\binom{3}{2}(\frac{1}{M})^2(\frac{M-1}{M})$ denotes the probability that, in the three threads that have already been suspended, one thread remains suspended and the two other threads are reactivated.

Having modeled the transition probabilities, we construct the transition matrix M_T and find the steady state probability distribution vector by: $\overrightarrow{v_s} = [\mathrm{prob}(S_0)\ \mathrm{prob}(S_1)\ \ldots\ \mathrm{prob}(S_N)] = \lim_{n\rightarrow\infty} \overrightarrow{v_i} M_T^n = \overrightarrow{v_s} M_T$ where $M_T = [p_{ij}]$ is an $(N+1)$-by-$(N+1)$ transition matrix, created using p_{ij} defined in Table 1, and $\overrightarrow{v_i} = [1\ 0\ \ldots\ 0]$ is the initial state vector (initially, all N threads are ready to issue an instruction). The equation for $\overrightarrow{v_s}$ can be evaluated quickly using standard Markov chain analysis [10].

3.4.2 Heterogeneous Workloads and Multiple Stall Conditions

When the threads running on a fine-grained multithreaded core are different, the probability of being suspended for each thread is different. Moreover, there are usually multiple types of events that can stall a thread and these events often have different latencies.

One way to extend the model described in Section 3.4.1 for heterogeneous workloads is to separate each state of the model into multiple states. For example, rather than using a single state S_1 to represent that the number of suspended threads is one, we can use N states to specify which particular thread out of all N threads is suspended. Then different probabilities p for each thread are used for the transitions from S_0 to one of each of these states. To completely specify which threads are suspended in all situations, we need 2^N states, making the Markov model more complicated.

To tackle the challenge of different thread types without introducing extra states, we instead compute a single value $\overline{p}$ for all the threads as follows: $\overline{p} = \sum_{i=1}^{N} p_i w_i$, where $\overline{p}$ denotes the average probability that a thread will be suspended, p_i is the probability that thread i will be suspended, and w_i is a weight factor equal to the number of instructions executed for the thread, divided by the total number of instructions executed on the core. Therefore, $\overline{p}$ can be further expressed as: $\overline{p} = \sum_{i=1}^{N} \frac{I_i}{N_i} \frac{N_i}{N_{total}} = \sum_{i=1}^{N} \frac{I_i}{N_{total}}$, where I_i is the number of instructions causing the thread i to be suspended, N_i is the number of instructions executed for thread i, and N_{total} is the total number of instructions executed on the fine-grained multithreaded core.

Given the total number of instructions executed on a core, we first assume that the number of instructions exe-

Table 2. Simulation Parameters

Number of Cores	8 in-order cores
Number of Threads	32, 4 threads per core
Pipeline	6 stages
Branch Predictor	4Kb gShare, 256-entry BTB, private
L1 Instruction Cache	16KB, 16B/line, 4-way, LRU, 1-cyc lat., private
L1 Data Cache	16KB, 16B/line, 4-way, LRU, 1-cyc lat., private
L2 Unified Cache	3MB, 64B/line, 12-way, LRU, 10-cyc lat., global
L1-L2 Interconnection	Crossbar [15]
ITLB/DTLB	64 entry, private
Memory Latency	110 cycles

cuted by each thread on that core (i.e., $\mathtt{N_i}$) is proportional to the thread's isolated IPC computed using a simple sum of CPI component model [9]. Then, based upon the estimated $\mathtt{N_i}$ for each thread, we use the novel cache contention model described in Section 2.3 to predict its number of extra cache misses due to cache sharing. Next, we add the product of the number of extra misses for each thread and the miss latency to the original number of cycles taken to execute $\mathtt{N_i}$ instructions and then divide the sum by $\mathtt{N_i}$ to estimate a new CPI for the thread. Finally, we update $\mathtt{N_i}$ for each thread to make it proportional to its new CPI and use the updated $\mathtt{N_i}$ to obtain $\overline{\mathtt{p}}$ for the Markov chain model.

Next, we tackle the issue of modeling systems where multiple events may suspend a thread and where each has a distinct latency. Similar to finding an average $\mathtt{p}$ for all threads, we compute an average $\mathtt{M}$ for all the events that might suspend a thread as follows: $\overline{\mathtt{M}} = \sum_{\mathtt{i}=1}^{\mathtt{k}} \mathtt{M_i s_i}$, where $\mathtt{k}$, $\mathtt{M_i}$, $\mathtt{s_i}$ represent the number of different types of events, latency of an event $\mathtt{i}$, and the weight of the event, respectively. The latency of each event suspending a thread is assumed to be known (a simplification in the case of DRAM and interconnect contention). To find the weight of an event, we also need to know how many times the event occurs during execution. The frequency of long latency floating point instructions can be obtained from the instruction trace used during circular sequence profiling. To find the number of cache misses, the cache contention model described in Section 2.3 is used. Above, the weight $\mathtt{s_i}$ for an event is computed as the product of its number of occurrences and its latency, normalized as a fraction of total execution time. For example, if $\mathtt{Event_1}$ with 5-cycle latency happens 50 times and $\mathtt{Event_2}$ with 100-cycle latency happens 10 times, then the average $\mathtt{M}$ is calculated as $\overline{\mathtt{M}} = 5 \times \frac{50\times5}{50\times5+10\times100} + 100 \times \frac{10\times100}{50\times5+10\times100} = 81$.

Although our Markov chain model is proposed for modeling fine-grained multithreaded architectures, we believe that, by considering state transitions to occur only for long latency L2 cache misses (rather than L1 misses) and scaling the estimated throughput by a factor taking account of the peak issue width of an SMT core (assuming a well balanced design [12]) and the memory level parallelism within a thread, our model can potentially provide performance predictions for more complex processor cores. However, a detailed evaluation of this extension is beyond the scope of this paper.

Table 3. Simulated Workloads for Each Core

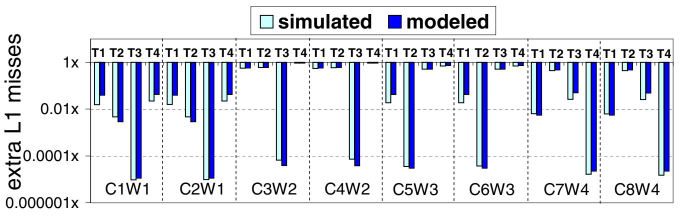
Core 1 Workload 1 (C1W1)	ammp-applu-art-mcf
Core 2 Workload 1 (C2W1)	ammp-applu-art-mcf
Core 3 Workload 2 (C3W2)	bzip2-mgrid-swim-equake
Core 4 Workload 2 (C4W2)	bzip2-mgrid-swim-equake
Core 5 Workload 3 (C5W3)	ammp-art-mgrid-equake
Core 6 Workload 3 (C6W3)	ammp-art-mgrid-equake
Core 7 Workload 4 (C7W4)	applu-bzip2-mcf-swim
Core 8 Workload 4 (C8W4)	applu-bzip2-mcf-swim

4 Methodology

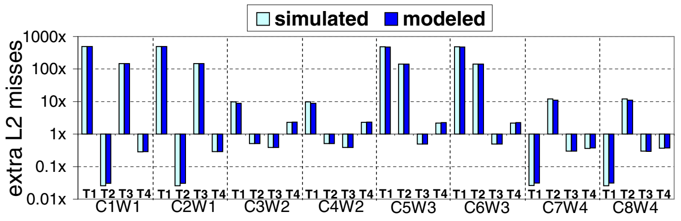
We evaluated the accuracy of our analytical models in three steps. First, we compared our analytical models against a detailed simulator for a microarchitecture similar to Sun Microsystems' Niagara T1 [13] that we developed by modifying SMTSIM [24]. Table 2 shows the microarchitectural parameters simulated. As discussed earlier in Section 3.4, our baseline model allows at most one outstanding miss per thread. We do not enforce cache inclusion between the L1 and the L2 cache (earlier cache contention studies have shown this has minimal effect [3]). Second, we applied our combined analytical cache contention and Markov chain throughput model to obtain two optimized application-specific multithreaded processor designs. Finally, we validated our analytical models by comparing their predictions against a Sun Fire T1000, which has a T1 Niagara processor containing 8 cores each with 4 thread contexts (32 threads total) and runs the Solaris 10 operating system. We used Shade [6] to collect instruction traces that are later analyzed to obtain inputs to our models. We report the arithmetic mean of the absolute value of error since it reports the largest error number and since we are interested in averaging the *error* of the IPC prediction (not the average speedup), but we also report geometric mean and harmonic mean of the absolute error to allay any concern that these numbers might lead to different conclusions.

When we validated our cache contention model, we used our detailed simulator and hardware performance counters to obtain information about the number of instructions executed by each thread. Then for each thread, we analyzed its memory instructions trace to obtain the two temporal locality metrics that we proposed in Section 2.2 as well as the circular sequence profile. Next, we predicted the number of extra misses of each thread using the cache contention model described in Section 2.3 and compared the modeled extra misses to the result from our detailed simulator and performance counters. To obtain the actual number of extra misses, we also ran each thread alone to obtain the number of cache misses without cache sharing.

When we validated our Markov chain throughput model, we first approximated the number of instructions executed by each thread as proportional to its isolated IPC (we used performance counters to obtain the isolated IPC when we compared to hardware), given the total number of instructions executed by all threads in a core. Then we applied our cache contention model to predict the number of extra



(a) L1 data cache



(b) L2 cache

Figure 6. Ratio of the number of extra L1 and L2 cache misses due to cache contention to the number of misses when each thread runs alone. The order of the threads in each core is shown in Table 3.

misses for each thread and adjusted each thread's IPC by using the product of the number of extra misses and the miss latency. Then we re-estimated the number of instructions executed by each thread based on the thread's refined IPC. Next we applied the cache contention model again to approximate the extra number of misses for each thread based on its refined instruction count. Finally, we used the refined instruction count and the extra number of misses for each thread as the input to our Markov chain model to estimate the throughput.

For the comparison against the detailed simulator, we chose 8 benchmarks from the SPEC 2000 Benchmark Suite [22] to form heterogeneous multiprogrammed workloads for each fine-grained multithreaded core being simulated. Table 3 shows the simulated workload of each core for our first study. We ran our detailed simulator such that each core executed at least 400 million instructions.

For our hardware comparison, we evaluated homogeneous workloads consisting of a varying number of threads, each running a memory intensive application (mcf). We also evaluated several heterogeneous workloads consisting of multiple instances of two types of applications to obtain 32 threads in total running concurrently. Specifically, we consider workloads consisting of the following combinations: 16gzip+16eqk (each core runs two gzip and two equake), 16mcf+16gzip, 16mcf+16art. We compiled these benchmarks on the Sun Fire T1000 using gcc 3.4.3 with -O2 optimization and ran them with their train inputs (mcf and art) and test inputs (gzip and equake).

To obtain miss latencies required by our throughput model, we created several simple (about 20 lines in C) microbenchmarks, which we ran on a single thread context. Based upon these separate calibration measurements, we used fixed latencies of 20 cycles, 220 cycles, and 50 cycles for an L1 cache miss that hits in the L2 cache, an L2 cache miss, and a floating point instruction, respectively (these are higher than reported in [20]).

5 Experimental Results

In this section we evaluate our analytical models against both a cycle-accurate performance simulator and real hardware.

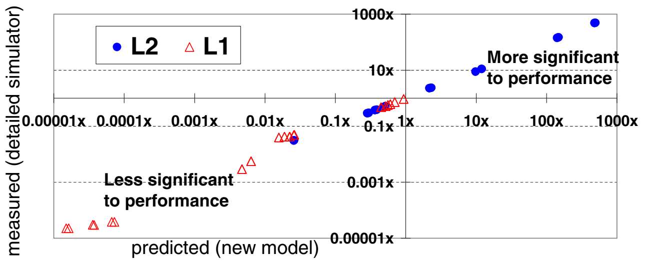


Figure 7. Scatter plot for the predicted and the simulated cache misses increase pair

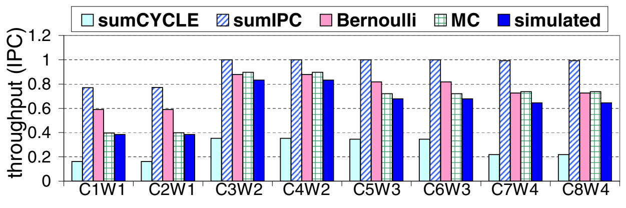
5.1 Model Accuracy Evaluation

Figure 6(a) and 6(b) illustrate the impact of cache sharing on the number of extra misses for our L1 data cache and L2 cache, respectively, when the heterogeneous workloads described in Table 3 are simulated. For each thread, we plot the ratio of the number of extra misses due to cache contention to the number of misses without cache contention (i.e., when the thread runs alone).

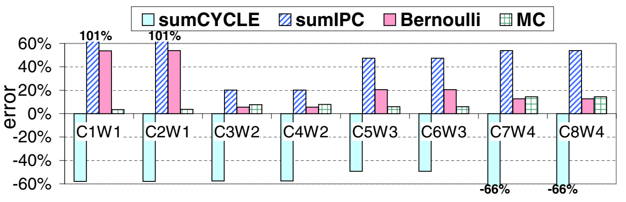
Figure 6(a) and 6(b) show that our cache contention model accurately predicts the number of additional cache misses due to cache sharing across varying workloads. For the L1 cache, the arithmetic mean of absolute error of our model is 40.3%, geometric mean is 12.1%, harmonic mean is 0.9%, and the maximum absolute error is 150.0%. For the L2 cache, the arithmetic mean of absolute error of our model is 4.4%, geometric mean is 1.1%, harmonic mean is 0.2%, and the maximum absolute error is 20.3%.

Figure 7 illustrates a scatter plot for the predicted and the simulated ratio pair. Each data point on this figure corresponds to a pair of bars in Figure 6(a) and Figure 6(b). From Figure 7 we notice that our predicted values are highly correlated with the simulated ones. The correlation coefficient between the predicted and the simulated ratio for L1 and L2 cache is 0.9993 and 0.9999, respectively.

We noticed that the average and maximum error for the L1 cache is higher than the L2 cache and we found it is mainly due to the fact that the number of extra L1 misses due to cache contention is much smaller than the number of extra L2 misses. For example, the maximum error of our model that predicts extra L1 misses is from T1 on Core 1, where the number of L1 misses is about 6153K without cache contention and it is increased by only 97K due



(a) Throughput (IPC)



(b) Error of different models

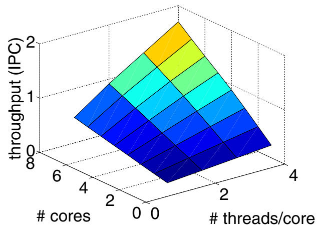
Figure 8. Predicted throughput and error from different models compared to the simulated throughput

to cache contention. For this thread, our model predicts the number of extra misses to be 242K, resulting a 150% error. However, this error does not noticeably impact the accuracy of the Markov chain throughput model since the total number of extra L1 misses is quite small compared to the number of L1 misses without cache contention (i.e., the number of L1 misses increases only by 1.6%).

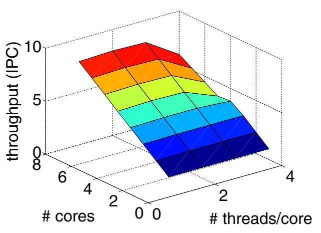
Figure 8(a) compares the throughput on each core predicted by the different models described in Section 3 to the simulated throughput and Figure 8(b) shows the error of each model. The first bar ("sumCYCLE") corresponds to the model described in Section 3.1 that always underestimates the actual throughput (with an arithmetic mean of the absolute error of 58%) since the model assumes that the execution of each thread on a core is completely serialized. The second bar ("sumIPC") is obtained using the model described in Section 3.2 that always overestimates the actual throughput (with an error of 55% on average). The third bar ("Bernoulli") represents the Bernoulli model described in Section 3.3 that is more accurate than the previous two models (with an average error of 23%), but its error is still significant for some workloads (e.g., the error is 54% for Core 1 Workload 1) since it does not take into account cache contention. Finally, the fourth bar ("MC") represents the Markov chain model proposed in Section 3.4 combined with our cache contention model proposed in Section 2.3 that predicts the throughput significantly more accurately with an arithmetic mean of the absolute error over eight cores of 7.9% and a maximum error of 14.3%.

5.2 Case Study: Optimizing Threads Per Core for Application-Specific Workloads

In this section, we present a case study of using our analytical models to predict the overall throughput of systems with different number of cores and threads with the aim



(a) mcf



(b) bzip2

Figure 9. Optimization Case Study: Modeled throughput under different configurations

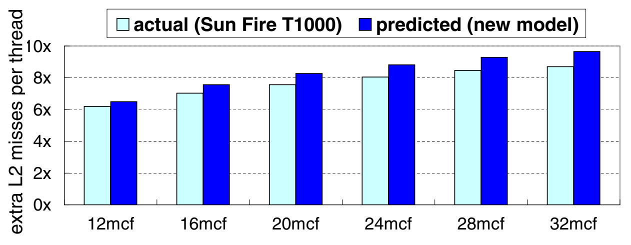
of finding the optimal configuration for some application-specific workloads. We choose *mcf* and *bzip2* for the study.

We vary the number of cores of the system from 1 to 8 and the number of threads per core from 1 to 4 (i.e., 32 configurations in total) and in each configuration we predict the overall throughput when running the same application for each thread. Figure 9(a) and 9(b) show the modeled throughput in all configurations for *mcf* and *bzip2*, respectively.

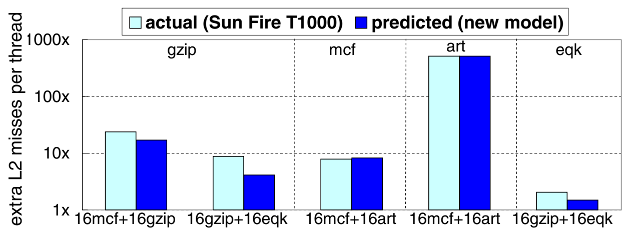
It is interesting to note that our analytical model indicates that when the number of cores is fixed, the throughput scales up as the number of threads per core increases for *mcf* but down for *bzip2*. The reason for this is the memory-bound nature of *mcf*, which causes the throughput to be relatively low when *mcf* runs by itself on a core. Thus, when running and sharing resources with other threads, the overall throughput improves. On the other hand, *bzip2* is compute-bound and, when the number of threads sharing a core is over two, extra cache misses due to cache contention actually outweigh the advantage of fine-grained multithreading, degrading the overall throughput. Therefore, the optimal configuration for *mcf* is a system with eight cores with four threads per core, while the optimal configuration for *bzip2* is eight cores with two threads per core. Our analytical model predicts the same optimal configuration for each workload as the one obtained from detailed simulations.

For *mcf*, the arithmetic mean of the absolute error, the geometric mean, and the harmonic mean are 2.2%, 2.1%, and 1.8%, respectively. For *bzip2*, they are 14.3%, 4.0%, and 1.0%, respectively. Moreover, our model is much faster than running detailed simulations. The time taken by our analytical model to find the throughput in all 32 configurations for each benchmark was as follows: We required 20 minutes to collect traces and statistics required for the cache contention model and roughly 10 seconds to evaluate the cache contention and Markov chain model, while the detailed simulator takes more than 22 hours (i.e., overall, 65 times faster[4]).

[4]In some sense we are conservative in stating the speedup of our analytical models over detailed simulations as traces only need to be collected and analyzed once for each application and the overhead can be amortized by reusing them.



(a) homogeneous workloads



(b) heterogeneous workloads

Figure 10. Predicted ratio of the number of extra L2 misses (per thread) due to cache contention to the number of L2 misses when a thread runs alone compared to an actual hardware (Sun Fire T1000)

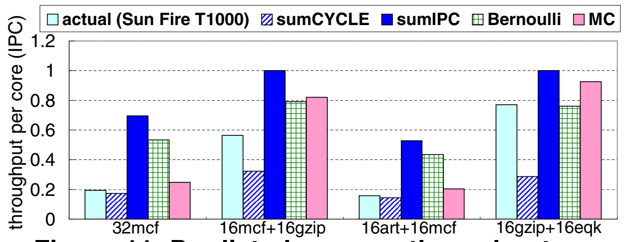


Figure 11. Predicted average throughput per core versus the throughput reported by performance counters on a Sun Fire T1000

5.3 Hardware Validation

While our detailed simulator captures the effects we set out to model analytically, we wish to find out how closely our models predict real hardware performance. Thus, we compare our models to the actual performance measured on a Sun Fire T1000 server (i.e., a Niagara T1 machine [13]). Figure 10(a) compares the predicted against actual increase in L2 load misses per thread, normalized to the number of L2 cache load misses in the absence of contention for a homogeneous workload—a varying number of copies of the memory intensive benchmark *mcf*. From Figure 10(a) we observe that as the number of threads running in parallel increases, the number of extra L2 load misses per thread also increases due to additional cache contention. Our model accurately predicts this trend. In Figure 10(a) the arithmetic mean of the absolute error of our model is 8.7% (improving the accuracy of the prior model [3]—not shown in the figure—by a factor of 48.1×). Similarly, Figure 10(b) compares the average number of extra L2 load misses per thread executing a variety of heterogeneous workloads. In Figure 10(b), the arithmetic mean of the absolute error of our model is 22.6% (improving the accuracy of the prior model [3]—not shown—by a factor of 163.4×).

Figure 11 compares the throughput per core predicted by our combined cache contention and Markov throughput model ("MC"), as well as that predicted by the three simpler models described in Sections 3.1 to 3.3 ("sumCYCLE", "sumIPC", and "Bernoulli", respectively) against the Sun Fire T1000 server ("actual (Sun Fire T1000)") for four different workloads (one homogeneous, three heterogeneous). We observe that MC tends to track the actual hardware better than the simpler models. The standard deviation of the error for MC is 10.7% which is much lower than 26.1%, 113.7%, and 90.8% for sumCYCLE, sumIPC, and Bernoulli, respectively. Moreover, we notice that MC always overestimates actual throughput (by 30.6% on average—arithmetic mean of the absolute error, 29.4% for geometric mean, or 28.2% for harmonic mean). We believe this bias is due partly to the fact that we used an average L2 cache miss latency measured in the absence of DRAM contention. The detailed performance simulator used for comparison in Section 5.1 and 5.2 does not model detailed DRAM timing (our detailed simulator also does not model page faults). Analytically modeling the increased L2 cache miss latency due to DRAM contention when multiple threads run together is beyond the scope of this study and is left for our future work.

6 Related Work

Yamamoto et al. [26] describe an analytical model for a *multistreamed* superscalar processor. The processor they model is essentially an SMT processor (capable of issuing instructions from multiple threads to the superscalar core on a single cycle) with the assumption of perfect branch prediction and caches that always hit. They present a probabilistic model to account for structural hazards by considering the instruction mix versus number and type of function units and use a correction factor computed by performing detailed simulation for a single thread in isolation to account for the effects of control and data hazards. In contrast, our model, while focused on single-issue fine-grained multithreaded architectures, requires no detailed simulation infrastructure to obtain accurate throughput predictions.

Thiebaut and Stone [23] propose an analytical model for cache-reload transients (i.e., cache contention due to context switching) and Agarwal et al. [2] propose an analytical cache model to predict a process' cache miss rate. Both do so on a time-sharing system running multiprogrammed workloads. Falsafi and Wood [7] extend Thiebaut and Stone's model to account for the interaction of more than two threads and apply analytical modeling to predict cost/performance of the Wisconsin Wind Tunnel simulator. They also consider the case where parallel threads share some memory regions (leading to prefetching effects), which we did not consider in our study. While all of the above work assumes a binomial distribution of cache blocks among cache sets, our model uses S(x) and b(i, x) to more directly quantify the probability of a co-scheduled thread inserting a certain number of accesses into a cache set to better model the very fine-grained interleaving of threads (we assume that threads are interleaving access by access).

As described in Section 2.1 the *Prob* model of Chandra et al. [3] is accurate if the number of threads is significantly below the associativity of the cache being shared, but does not model contention well as the number of threads is increased. While Chandra et al. mention their model could be applied to predict overall performance, they do not describe such a technique in detail and do not provide quantitative evaluation of throughput modeling as we have done in Section 3 and Section 5, respectively.

7 Conclusions

In this paper we present novel techniques for analytically modeling the throughput of fine-grained multithreaded chip multiprocessors such as Sun Microsystems' Niagara. Throughput is estimated by applying a Markov chain model where states represent numbers of stalled threads, along with transition statistics obtained using a novel cache contention model that takes account of temporal locality within a thread and accurately predicts cache contention among a large number of threads. Overall, our models achieve an accuracy of 7.9% on average versus a cycle-accurate simulator that captures the major effects we attempt to model analytically. Our models also find the same optimum configuration for some application-specific workloads as the detailed simulator. Furthermore, our models achieve these results 65 times faster than detailed performance simulations. We also validate our models against a Sun Fire T1000 and show that our models accurately predict the performance of the real hardware.

Acknowledgments

We would like to thank Greg Steffan for providing access to the Sun Fire T1000 machine we compared against in this study, as well as Lawrence Spracklen and Sun Microsystems for access to additional hardware and for help when we validated our models against the Sun Fire T1000. We thank Jeff Brown for providing us with the simulated binaries of SMTSIM. We also thank Ali Bakhoda, Wilson Fung, George Yuan, and the anonymous reviewers for their valuable comments on this paper. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] V. S. Adve and M. K. Vernon. Parallel Program Performance Prediction Using Deterministic Task Graph Analysis. *ACM TOCS*, 22(1):94–136, 2004.

[2] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. *ACM TOCS*, 7(2):184–215, 1989.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA-11*, pages 340–351, 2005.

[4] X. E. Chen and T. M. Aamodt. Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs. In *MICRO-41*, pages 59–70, 2008.

[5] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO-40*, pages 249–261, 2007.

[6] B. Cmelik and D. Keppel. Shade: A Fast Instruction-set Simulator for Execution Profiling. In *SIGMETRICS*, pages 128–137, 1994.

[7] B. Falsafi and D. A. Wood. Modeling Cost/Performance of A Parallel Computer Simulator. *ACM TOMACS*, 7(1):104–130, 1997.

[8] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. Toward A Multicore Architecture for Real-time Ray-tracing. In *MICRO-41*, pages 176–187, 2008.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann, 2007.

[10] P. G. Hoel, C. J. Stone, and S. C. Port. *Introduction to Stochastic Processes*. Waveland Press, 1987.

[11] T. S. Karkhanis and J. E. Smith. A First-order Superscalar Processor Model. In *ISCA-31*, pages 338–349, 2004.

[12] T. S. Karkhanis and J. E. Smith. Automated Design of Application Specific Superscalar Processors: An Analytical Approach. In *ISCA-34*, pages 402–411, 2007.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[14] P. Michaud, A. Seznec, and S. Jourdan. An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors. *IJPP*, 29(1):35–38, 2001.

[15] S. Microsystems. *OpenSPARC^{TM}T2 Core Microarchitecture Specification*. Sun Microsystems, Inc., 2007.

[16] D. B. Noonburg and J. P. Shen. Theoretical Modeling of Superscalar Processor Performance. In *MICRO-27*, pages 52–62, 1994.

[17] D. B. Noonburg and J. P. Shen. A Framework for Statistical Modeling of Superscalar Processor Performance. In *HPCA-3*, 1997.

[18] D. J. Ofelt. *Efficient Performance Prediction for Modern Microprocessors*. PhD thesis, Stanford University, 1999.

[19] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *SIGMETRICS*, pages 48–60, 1993.

[20] D. Sheahan. *Developing and Tuning Applications on UltraSPARC®T1 Chip Multithreading Systems*. Sun Microsystems, Inc., 1.2 edition, October 2007.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS-X*, pages 45–57, 2002.

[22] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.spec.org.

[23] D. Thiebaut and H. S. Stone. Footprints in the Cache. *ACM TOCS*, 5(4):305–329, 1987.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA-22*, pages 392–403, 1995.

[25] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovi. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, 2007.

[26] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirosky. Performance Estimation of Multistreamed, Superscalar Processors. In *27th Hawaii Int'l Conf. System Sciences*, Jan. 1994.

[27] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The Future of Simulation: A Field of Dreams. *Computer*, 39(11):22–29, 2006.

[28] L. Zhao, R. R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring Large-Scale CMP Architectures Using ManySim. *IEEE Micro*, 27(4):21–33, 2007.