Modeling the Effects of Memory Hierarchy Performance On Throughput of Multithreaded Processors

Alexandra Fedorova
Margo Seltzer
and
Michael D. Smith

TR-15-05

Computer Science Group
Harvard University
Cambridge, Massachusetts

Modeling the Effects of Memory Hierarchy Performance On Throughput of Multithreaded Processors

Alexandra Fedorova[†‡], Margo Seltzer[‡], Michael D. Smith[‡]

[†]*Sun Microsystems*, [‡]*Harvard University*

ABSTRACT

Understanding the relationship between the performance of the on-chip processor caches and the overall performance of the processor is critical for both hardware design and software program optimization. While this relationship is well understood for conventional processors, it is not understood for new multithreaded processors that hide a workload's memory latency by executing instructions from several threads in parallel. In this paper we present a model for estimating processor throughput as a function of the cache hierarchy performance. Our model has a closed-form solution, is robust against a range of workloads and input parameters, and gives estimates of processor throughput that are within 13% of measured values for heterogeneous workloads. We demonstrate how this model can be used in an operating system scheduler tailored for multithreaded processor systems.

1. INTRODUCTION

In this study we develop an analytical model of the effects of processor cache miss rates on the overall performance of a multithreaded processor. Multithreaded (MT) processors are designed to hide the effects of memory latency by running multiple instruction streams in parallel [5-10]. An MT processor has multiple thread contexts, and it interleaves execution of instructions from different threads. As a result, if one thread blocks on a memory access, other threads can make forward progress. The motivation for this architecture is to improve performance of an important class of modern memory-intensive applications, such as web services, application servers, and on-line transaction processing systems, that are notorious for causing frequent processor stalls and have processor pipeline utilizations of less than 20% [1, 2, 5, 21, 22]. While hardware multithreading is not a new idea, the first commercial systems equipped with multithreaded processors, such as Intel’s Hyper-threaded Pentium 4 [10] and IBM’s RS64 IV [29], have been made available only recently. This architecture has quickly become popular: the majority of new processors that are being released are multithreaded. Even so, multithreaded processor architectures are still evolving, and as they do their designs are becoming more complex. IBM has recently released its first *multithreaded chip multiprocessor* [25]; Sun Microsystems and Intel have plans to release similar processors in the fall of 2005 [26, 27]. Our lack of practical experience with multithreaded architectures suggests that we do not yet have a complete understanding of how these processors perform and how best to design them.

Analytical modeling, along with simulation, is a valuable tool in microarchitectural development and analysis. A model for processor performance allows efficient exploration of the design space. The amount of exploration available via simulation can be limited, because accurate simulations are time-consuming; analytical modeling does not have such limitations. Even though analytical modeling is usually less accurate than detailed simulation, it is useful for studies that explore how components in the design interact with one another and for evaluating hypothetical future designs where the lack of a design blueprint makes complete accuracy impossible [28].

In designing multithreaded processors, it is crucial to make the right tradeoff between the chip real estate that is used for cache and for thread hardware contexts. Having more hardware contexts increases latency-hiding capabilities. On the other hand, not having enough cache may cause memory latency to become so high that multithreading will not be able to hide it. Our model estimates the amount of latency that a multithreaded processor can hide, depending on its cache size and the number of hardware contexts, and can be used in studying the effects of such tradeoffs.

Since hardware multithreading hides memory latency software designers may need to place less emphasis on optimizing their applications for high cache hit rates. It is, therefore, necessary to develop new intuition for the kind of cache performance that is acceptable for multithreaded processors. Having the model for estimating the impact of cache miss rate on processor performance aids in doing so.

In addition to providing a valuable tool for design-space exploration, in Section 7, we show how to use our model in the implementation of a scheduling algorithm tailored for multithreaded chip multiprocessors [20].

Our model estimates processor throughput as a function of cache hierarchy performance. As a metric for processor throughput we use *instructions per cycle* (IPC). We chose this metric because it is important for long-running throughput-oriented workloads such as application servers and databases, whose performance is often expressed as the sustained number of requests per second – the quantity that is ultimately linked to processor IPC. To capture the cache hierarchy performance, we use cache miss rate – the number of cache misses per instruction. Therefore, our model estimates *processor IPC for a given cache miss rate.*

The challenge in developing such a model for multithreaded processors, which is fundamentally different from developing a similar model for conventional processors or multiprocessors, lies in the fact that a multithreaded processor partially masks the memory latency experienced by the threads (See Figure 1). Modeling the extent of such masking is essential to producing an accurate model, and we have developed a powerful technique for doing this.

Another challenge we address is modeling the effects of memory bandwidth contention. When several threads issue requests to main memory simultaneously, they compete for memory bandwidth. A similar contention is present on multiprocessor systems. Previously proposed modeling techniques to address such contention were not appropriate for our model because they either produced inaccurate estimates or were too computationally expensive. We have developed a new technique for modeling memory-bus delays, which produces estimates that are on average within 12% of the measured values. Because this technique targets our specific problem domain, it is simple and has a closed-form solution.

Our IPC model estimates processor IPC to within about 13% of the measured values and has a closed-form solution. In the cases where the model produces errors, the errors are consistent in magnitude and direction, which indicates that the model is successful in predicting performance trends even when the actual IPC cannot be predicted with precision. The strength of our model comes from the fact that it is a function of how the processor hides memory latency, which is independent of cache architecture and workload characteristics. Because we developed a powerful technique for modeling this effect, our model works with a wide range of parameters and workloads.

The rest of this paper is organized as follows: In Section 2 we describe the multithreaded processor whose behavior we model, and the methodology. In Section 3 we introduce some terminology and describe a base model for memory latency for single-threaded workloads. In Section 4 we present our technique for modeling how a multithreaded processor hides single-thread memory latencies. In Section 5, we describe how we modeled delays associated with competition for memory bandwidth. In Section 6, we put together the pieces of our model, validate it and discuss its strengths and weaknesses. In Section 7, we demonstrate how our model can be used in practice. We discuss related work in Section 8 and conclude in Section 9.

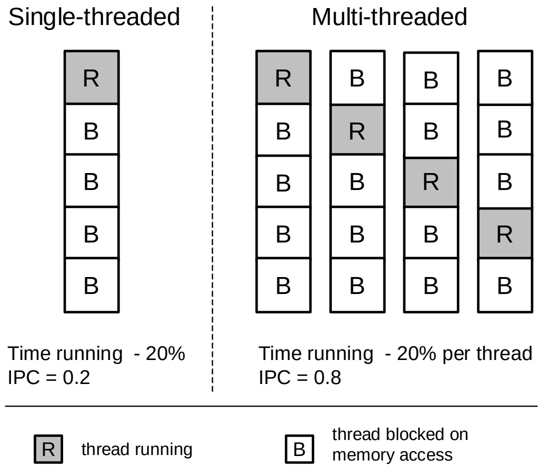

Figure 1. Each box denotes the state of the processor pipeline for a single cycle. For a single-threaded processor, if a thread spends 20% of its time running and the remainder of its time blocked handling cache misses, the processor is blocked 80% of the time and completes only one instruction in five cycles, yielding IPC of 0.2. A multithreaded processor hides memory latency. Although each thread spends 80% of the time in the blocked state, overall, the processor is blocked only 20% of time, yielding IPC of 0.8

2. SYSTEM AND METHODOLOGY

2.1. Multithreaded Processor

For this study, we collect data and perform validation experiments on a simulated machine. In contrast to using a real machine, this gives us freedom to experiment with a variety of machine configurations (i.e., cache size, memory bandwidth). In this section we describe the processor that we simulate and the simulator itself.

The architecture of our simulated multithreaded processor is based on fine-grained multithreading (interleaving), proposed by Laudon et al. [6]. The processor has several hardware thread contexts, where each context consists of a set of registers and other thread state. The processor interleaves execution of instructions from the threads, switching between contexts on each cycle in a round-robin fashion. When one or more threads are blocked, the processor continues to switch among the remaining available threads. If there is not a thread that is ready to issue an instruction, the processor stalls, waiting for some thread to become ready.

We model a simple RISC pipeline with one set of functional units (i.e., arithmetic logical unit, instruction fetch unit, etc.). We decided to simulate a simple, classical RISC processor, as opposed to a complex out-of-order processor, because we believe that this is a viable architecture for future MT processors. A simple pipeline occupies less space and allows for placing more hardware contexts on a chip; a previous study showed that for transaction-style workloads, pipeline complexity should be traded off for increased number of hardware contexts [11]. Additionally, we believe that the results of our study are applicable to a wide range of multithreaded architectures, because, as we will show, our model decouples the performance of the memory hierarchy from the performance of the processor pipeline.

For the purposes of validating our model, we use an MT system simulator [12], built on top of the Simics simulator of the UltraSPARC II® processor [13]. Simics can bootstrap the simulated machine with the Solaris™ operating system and standard Unix environment. All the simulations described in this paper are execution-driven and include both user-level and OS code.

The simulator accurately simulates pipeline contention, the L1 cache, bandwidth limits on crossbar connections between the

L1 and L2 caches, the L2 cache, and bandwidth limits on the path between the L2 cache and memory. The processor is configured with four hardware contexts (this configuration has been shown to perform best with our workload [6]), a write buffer, an 8KB L1 data cache, a 16KB L1 instruction cache (both 4-way set-associative) and a unified 12-way set-associative write-back L2 cache, whose size we vary depending on the experiment. We chose cache sizes to be similar to those used in the hyper-threaded Pentium 4, a multithreaded processor that is commercially available at the time of this writing [10].

2.2. Methodology

In a previous study we found that multithreaded processors are able to effectively hide latency from faults in the L1 cache, however poor performance in the L2 can adversely affect processor IPC [20]. This implies that variation in the L2 miss rates produces greater variation of processor IPC than variation in the L1 miss rates. In this study we model the effects of L2 miss rates on processor IPC, because this way we are able to validate our model for a wider range of IPCs. However, there is nothing in our model that precludes it from being used for other levels of the cache hierarchy, such as L1 or L3 caches.

To develop and validate our model, we use the SPEC CPU 2000 benchmark suite. These benchmarks are appropriate for studies of memory hierarchy, because this benchmark suite has been improved from previous versions to include programs whose memory footprints are much larger than traditional cache sizes [17]. We experiment only with integer benchmarks, because our simulator does not simulate contention for the floating-point unit. To ensure that our model works for multiple workloads, we train and test our model using distinct sets of benchmarks. To validate that our model is robust against a range of input parameters, we use several L2 cache sizes, ranging from an unrealistically small 48KB to a more realistic 192 KB. Development of our model followed a three-step process: first we modeled the L2-miss latency for a single-threaded workload. Then, to estimate the effect of such latency for the multithreaded workload, we developed a technique for estimating how a multithreaded processor hides individual threads' latencies. Next, so that we could validate our model using a realistic machine configuration, we developed a method to estimate delays associated with contention for the memory-bus. We describe these three pieces of our model in Sections 3, 4 and 5, respectively.

3. MODEL PRIMER

We introduce our model by stating our assumptions and demonstrating how we model L2-miss latencies for a single-threaded workload. Our ultimate goal is to model how the processor hides individual threads' latencies, but as a prerequisite we need to be able to estimate them.

3.1. Assumptions

The performance of the L2 cache is only one factor that affects processor IPC. Processor IPC is determined by a multitude of other factors, such as the architecture of processor pipeline, the instruction mix of the workload, and the performance in the L1 caches. Modeling the effects of these factors is outside the scope of this work. Our goal is to study the effects of the L2 performance on IPC in an isolated fashion. In order to do this, we assume the knowledge of the *ideal IPC* – the IPC that the workload has when there are no capacity or conflict misses in the L2. In other words, a workload experiences its ideal IPC when it runs with an infinitely large L2 cache. Using this assumption is a standard approach when modeling the effects of memory hierarchy on the overall performance [18, 19].

To develop our model, we assume that we can measure the L2 read-miss rate and the L2 write-miss rate. The read-miss rate includes both data and instruction misses. These quantities can be measured by reading hardware performance counters usually available on modern processors.

3.2 Single-threaded workload model

We begin by modeling the L2-miss latency for a single-threaded workload, so that later we could model how a multithreaded processor hides such single-thread latencies. The model estimates latency *per instruction*: how many cycles per instruction a thread spends handling L2 misses.

First let us introduce some definitions:

L2_CPI – Per-instruction L2-miss latency. For a given L2 miss rate, the number of cycles per instruction that a thread is blocked handling misses in the L2 cache.

Ideal_CPI – the inverse of ideal IPC (recall section 3.1);

CPI – cycles per instruction given some L2 miss rate;

L2_MR – Number of L2 misses per instruction – this includes the read-miss rate and the write-miss rate.

L2_MCOST – the cost, in cycles, of handling each miss in the L2 cache. This is the cost of going to memory from the L2 cache[1] (we set it to 120 cycles). In this section we assume that the bandwidth between the L2 and main memory is infinite, so there is no delay associated with waiting for the memory bus.

L2_CPI depends on the L2 miss rate and the cost of each miss:

$$L2\_CPI = L2\_MR * L2\_MCOST \tag{1}$$

We also observe that the thread's CPI is comprised of its ideal CPI and the CPI that is due to handling the misses in the L2:

$$CPI = Ideal\_CPI + L2\_CPI \tag{2}$$

[1] If a workload is multithreaded, there is also a cost associated with waiting for a cache line if it is in use when the thread accesses it. We observed from our data that this cost does not have a high impact on performance, so we disregard it altogether. Neither do we account for communication costs associated with a cache consistency protocol. This would be relevant for systems with multiple caches on the same level of the memory hierarchy. A model for such costs has been described elsewhere [18] and can be easily incorporated into our model if needed.

In the simplest case, we use equation (1) to compute the L2-miss latency given a particular L2 cache miss rate. This approach works if we have a write-though cache and a processor that is not equipped with a write buffer. However, we model a more sophisticated system with a write-back cache and a write buffer, where this approach does not work for the following reasons: Writing in the cache creates dirty cache lines that need to be written back: this may increase the cost a of a cache miss. Write buffer absorbs the write misses, making them non-blocking. Therefore, a write miss does not necessarily stall the thread. We explain how we account for the effects of write-backs and the write buffer in the following sub-sections.

3.2.1. Effect of write-back transactions

A write-back transaction occurs whenever the cache needs to evict a dirty cache line. For example, if a read transaction misses in the cache and the cache line that it needs to use is dirty, this line will be written to memory before it is used. In this case, a read transaction has to pay an additional penalty of *L2_MCOST*.

Therefore, in order to fully account for all memory-access penalties, the L2 miss rate must include write-back transactions. We assume that the rate of write-back transactions is known to us, i.e., we can measure it by reading hardware counters. However, for situations when the write-back rate cannot be directly measured, we developed a way to estimate it.

Intuitively, the write back rate depends on the write miss rate, because it is the write misses that create dirty cache lines. We used linear regression analysis to analyze the relationship between the write-miss rate and the write-back rate, and obtained a linear model with a good fit. Using this model, it is possible to estimate the write-back rate of a workload to within 22% of actual values.

In the rest of this paper, when we talk about the L2 miss rate, we include the write-back rate, thereby accounting for all cache transactions that result in memory-access penalty.

As will become evident in the next section, for our model we need to distinguish between the read miss rate and the write miss rate. Therefore, we need to know which fraction of the write-back rate should be included in the read-miss rate, and which should be included in the write-miss rate. As one could expect, it turns out that this fraction is proportional to the fraction that the read- or write- miss rate contributes to the overall miss rate. For example, if read misses constitute 60% of all misses, then about 60% of all write-backs are triggered by read-miss transactions, and so 60% of the write-back rate should be included in the read-miss rate.

3.2.2. Write buffer effects

A write buffer cushions the effect of write misses: when a thread performs a write, the value is placed into the buffer, and the transaction completes immediately. The written values are propagated through the memory hierarchy asynchronously, without stalling the thread. The only time a write can stall the thread is when the write buffer becomes full – in this case the thread waits until some space becomes available in the write buffer.

In our simulated system, we model a write buffer with eight double-word entries. The write buffer is shared among threads that run on the same processor and is positioned above the first-level cache hierarchy. Our first-level cache is non-write-allocate: it does not allocate space in the cache in the event of a write miss. Therefore, all writes from the write buffer go directly to the L2 cache. As a result, the L2 performance affects the likelihood of the write buffer stalling the processor.

Queuing theory provides a natural approach to modeling the effect of a write buffer: a write buffer can be modeled as a server, and threads that send write requests as customers. Using a closed-network queuing model with limited buffer size, it is possible to estimate the delay associated with the buffer filling up. However, because solving such models is computationally expensive [18, 23, 24] (and we wanted our model to be suitable for on-line deployment), and because our model showed little sensitivity to this effect, we decided to use the following simplified approach.

We estimate the fraction of L2 write misses that are not absorbed by the write buffer and eventually stall the processor. This quantity depends on the write miss rate that a workload generates: the more writes that miss in the cache the longer it takes for the write buffer to propagate the writes and the more likely it is to fill up.

For our simulated architecture, if a workload has a writes-per-cycle rate of roughly 6,000 per million cycles or greater, about 90% of the L2 write misses stall the processor. At any rate less than that – only about 5% of L2 write misses stall the processor. We can use these numbers to augment Eq.1 as follows.

1. Split the L2 miss rate (*L2_MR*) into two parts: L2 read miss rate (*L2_RMR*) and L2 write miss rate (*L2_WMR*). The read miss rate includes both data reads and instruction fetches. Equation 2 now becomes:

$$L2\_CPI = (L2\_RMR + L2\_WMR) * L2\_MCOST \quad (3)$$

2. For workloads whose writes-per-cycle rate is above 6,000 per million cycles, multiply *L2_WMR* by 0.9 – to reflect that for such workloads about 90% of L2 write misses stall the processor. Similarly, for workloads whose writes-per-cycle rate is lower, multiply *L2_WMR* by 0.05. Let us call this multiplier *WMM* – the write miss multiplier. Equation 3 now becomes:

$$L2\_CPI = (L2\_RMR + L2\_WMR * WMM) * L2\_MCOST \quad (4)$$

Our approach to modeling the effect of write misses is architecture-dependent. In order to derive the *WMM* coefficient for a particular architecture it is necessary to characterize the effect of write miss rate on the write buffer on that architecture. In order to develop an architecture-independent method a more general approach needs to be used. Since accurately modeling this effect was not critical to the overall accuracy of our model, and was not the focus of this study, our approach suffices.

3.2.3. Evaluation

To train and test our model we separated the SPEC CPU 2000 integer benchmark suite in two groups[2]. The training set contained 164.gzip, 175.vpr-place, 175.vpr-route, 176.gcc, 186.crafty, 197.parser, and 255.vortex. The test set contained 181.mcf, 252.eon-cook, 254,gap, 256.bzip2 and 300.twolf.

To obtain *ideal_CPI* for these benchmarks, we simulated each one on a machine configured with a large L2 cache (3 MB), and measured the resulting CPI. (To perform a simulation, we fast-forward the execution past the initialization phase, and then perform a detailed measured simulation for 100 million instructions.)

To evaluate the model for single-thread L2-miss latency, we simulated the benchmarks in the test set on a machine with reduced L2 cache sizes (48KB, 96KB, 192KB), measured the CPI and the L2 miss rate. We use the L2 miss rate and the *ideal_CPI* to estimate the *CPI* using Eq.2 and 4. Our estimates were within 1% of the CPI measured during the simulation. Such high accuracy is not surprising because we model a simple and well-understood behavior using straightforward techniques. In the next section we explain how we tackled a more complicated problem.

4. MODELING LATENCY-HIDING

When a processor is executing a single-threaded workload, all cycles spent handling L2 misses stall the processor. Recall from Figure 1, however, that with a multithreaded workload, the processor hides the memory-access latencies of individual threads by running the threads in parallel. While a particular thread may be stalled on a cache miss, the processor could still be running, because there may be other threads that are not stalled. Therefore, only a fraction of all cycles spent handling L2 misses by individual threads stall the processor. In this section we show how to model this latency-masking effect by using the knowledge of how much time each individual thread stalls, and figuring out how this stall time overlaps with the non-stall time of the other threads. Then, we are able to estimate the effect of L2 miss rate on processor IPC.

4.1. The model

The key to understanding our representation of how the processor masks individual threads' memory latencies is the notion of the *probability that an individual thread is blocked on an L2 cache miss*. We refer to it as *thread-block probability.* In Section 4.1.1, we explain how we derive it. In Section 4.1.2, we show how to estimate the processor IPC based on it.

4.1.1. Thread-block probability

We derive a thread-block probability by examining how the thread spends its cycles during execution (recall Eq.2). *Ideal_CPI* gives us the number of cycles (per instruction) that the thread spends doing useful work. *L2_CPI* (Eq.4) gives us the number of cycles that a thread spends handling L2 cache misses. From this, we can determine what fraction of all cycles the thread spends blocked, handling L2 misses – this is the thread-block probability.

Although we derive the probability from per-instruction quantities, we are not concerned with the fact that an instruction may require varying number of cycles to complete depending on its type and that some instructions may not stall at all. Our objective is to derive a rough probability of a thread being in the blocked state: if we were to look at a window of thread's time on a processor, what fraction of this time would the thread be blocked? The thread-blocked probability estimates this fraction.

While it is sufficient to use a thread's *ideal_CPI* to derive thread-blocked probability for a single-threaded workload, for the multithreaded workload we need to use the ideal CPI of the multithreaded workload. When threads share the processor pipeline, they sometimes have to wait for their turn to use it. Therefore, each thread runs more slowly than it would had it had the pipeline all to itself, and the *ideal_CPI* of an individual thread increases, reflecting this competition. The ideal CPI of a multithreaded workload, *ideal_CPI_mt*, is defined as the CPI of a multithreaded workload that it achieves under no conflict- or capacity-misses in the L2. We estimate *ideal_CPI_ind*, the ideal CPI that an individual thread achieves when it shares the processor with the other threads as follows:

$$ideal\_CPI\_ind = ideal\_CPI\_mt * M \qquad (5),$$

where *M* is the number of thread hardware contexts on the processor. To understand why this works, consider how threads share the pipeline on our simulated processor with four hardware contexts. The processor issues instructions from one thread at a time, switching between the threads on every cycle in a round-robin fashion. Therefore, for each cycle that a thread spends doing useful work, it has to spend three cycles waiting while other threads are using the processor. We reflect this by multiplying the *ideal_CPI_mt* by the number of thread contexts.

Ideal_CPI_ind gives us the number of cycles per instruction that each thread spends doing useful work in a multithreaded scenario. Knowing the L2 miss rate for the multi-threaded workload and assuming that all threads equally contribute to the overall miss rate, we can compute the thread-blocked probability for the multithreaded scenario (*prob_blocked_ind)*:

$$prob\_blocked\_ind = \frac{L2\_CPI}{ideal\_CPI\_ind + L2\_CPI} \qquad (6),$$

Note that in Eq.6 we use *L2_CPI* from Eq.4 without modification. If all threads equally contribute to the overall L2 miss rate, their individual misses-per-instruction are the same as the misses-per-instruction for the entire processor: each thread handles 1/Mth of all misses, but it also executes only 1/Mth of all instructions. We discuss the validity of the assumption that all threads equally contribute to the overall miss rate in Section 4.2.

[2] We did not use 253.perlbmk, because it is a multi-process benchmark, and we needed to have single-threaded benchmarks.

Next we show how to use *prob_blocked_ind* to estimate the processor IPC for a multithreaded workload.

4.1.2. Modeling multithreaded IPC

When modeling how the multithreaded processor hides individual-thread memory latencies, we assume that all threads have equal probabilities of being blocked, *prob_blocked_ind*. We discuss how this assumption affects our model in Section 4.2.

If *M* is the total number of threads on a multithreaded processor, when executing a multithreaded workload where each thread periodically blocks on an L2 cache miss, the processor can be in one of the following *M+1* states:

(0) All *M* threads are blocked, none are running;
(1) Exactly one thread is running – the rest are blocked;
(2) Exactly two threads are running – the rest are blocked;
…
(M) All threads are running – none are blocked.

In State M, the processor is running at IPC equal to *ideal_IPC_mt* (the inverse of *ideal_CPI_mt*). In State 0, the processor is running at IPC equal to zero: when all threads are blocked it is not completing any instructions. When some threads are running and some are blocked, the processor is running at some IPC that is less than *ideal_IPC_mt* – we will refer to this quantity as *N_IPC*, where *N* corresponds to the number of threads that are *running*. So, for example, on a machine with four threads, when exactly three threads are running and one is blocked, the processor is running at *3_IPC*. We will return to *N_IPC* later in Section 4.1.2.2.

Knowing the probability that an individual thread is blocked, and defining the corresponding probability that a thread is running (*prob_running_ind)* as:

prob_running_ind = 1 – prob_blocked_ind,

we can compute probabilities *P* that a processor is in any of the states described above as follows:

$$P(i) = \binom{M}{i} * prob\_running\_ind^{i} * prob\_blocked\_ind^{M-i} \ ,$$

where *i* is the number of threads that are running in this state. Then, we can compute the IPC of a multithreaded workload for a given L2 miss rate (*IPC_mt*) by multiplying the IPC achieved in each state by the probability of that state, and summing across all states, as follows:

$$IPC\_mt = \sum_{i=0}^{M} P(i) * i\_IPC \qquad (7),$$

Using the model described by Eq.7 requires knowing *N_IPC*, and we have a way of deriving it, which we describe in Section 4.1.2.2. However, first, we want to evaluate the technique for modeling latency-masking in isolation. Therefore, we are going to measure *N_IPC*, and use it in Eq.7. We describe the details next.

4.1.2.1. Validating the model for latency-hiding

To validate the technique for modeling latency-hiding, we use the five SPEC benchmarks included in our test set (listed in Section 3.2.3). We create a multithreaded workload by running four copies of the same benchmark, so we have a total of five multithreaded benchmark groups. (We present the experiments with non-identical threads in Section 4.2.) We run the benchmarks by fast-forwarding the simulation past the initialization phase, and then performing the detailed simulation for 400 million instructions.

We obtain the *ideal_IPC_mt* (and *ideal_CPI_mt*) for each benchmark group by running it on a simulated processor configured with a large L2 cache (3MB). Since we have a total of four threads, we measure *N_IPC* for the values of one through three by running each benchmark group using one, two and three threads respectively.

Then, we simulate each benchmark group on machines configured with three reduced cache sizes (48KB, 96KB and 192KB) and measure the IPC and the L2 miss rate. We use the L2 miss rate, the *ideal_CPI_mt* and *N_IPC* to compute the estimated *IPC_mt* using Eq. 5, 6, and 7. We compare the estimated *IPC_mt* to the actual IPC that was measured during the simulations with reduced caches. We show how the actual IPCs compare to the estimated in Figure 2.

The estimated *IPC_mt* is on average within 3% of the actual IPC. The median error is 2%, and the largest is 8%. Also note that the estimated *IPC_mt* follows the same trend as the actual IPC. For example, for twolf, the actual IPC increases by 30% with each larger cache size, and the estimated *IPC_mt* has the same property. This is also the case for the other benchmarks.

These results suggest that our technique for modeling latency-hiding is accurate. The reason is that it is based on an intuitive representation of how individual threads' memory latencies overlap with one another. This representation does not depend on processor architecture or workload characteristics.

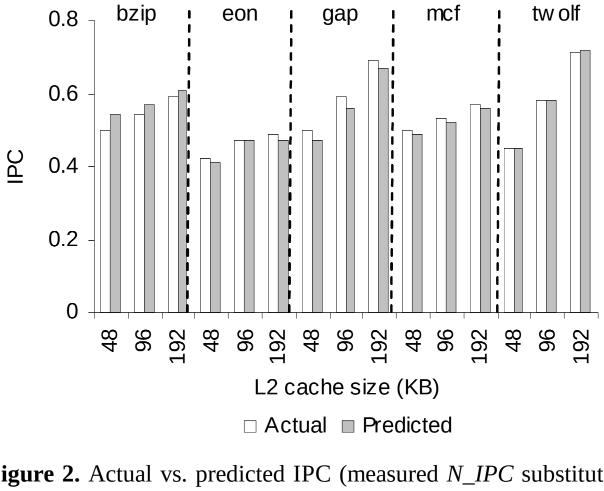

Figure 2. Actual vs. predicted IPC (measured *N_IPC* substituted in Eq.7).

However, to use the model in practice, we need to be able to derive *N_IPC*. We explain how to do this next.

4.1.2.2. Modeling N_IPC

We initially attempted to model *N_IPC* by scaling the ideal IPC in proportion to the fraction of a processor's thread contexts that were occupied. For example, to compute *3_IPC,* we multiplied the ideal IPC by ¾. Unfortunately, this simple approach consistently underestimates *N_IPC.* When some hardware contexts are left unused, the remaining threads are able to take advantage of the available resources, so this simple adjustment is not sufficient.

We observed that those thread groups whose *ideal_IPC_mt* is high ("fast threads") are better able to take advantage of free resources than thread groups whose *ideal_IPC_mt* is low ("slow threads"). Therefore, fast threads achieve the *N_IPC* that is closer to their ideal IPC than do slow threads. Thus we model *N_IPC* as a function of *ideal_IPC_mt* and the number of running threads (*N*). Fitting a linear equation using regression analysis resulted with a good fit (R-squared of 90%) and produced the following formula describing the relationship among these quantities:

$$N\_IPC = -0.69 + 0.2 * N + 0.94 * ideal\_IPC\_mt \qquad (8)$$

Figure 3 shows how the estimated *IPC_mt* compares to the actual IPC when, instead of using the measured *N_IPC* as in Figure 2, we used the *N_IPC* computed using equation 8. The estimations are still accurate – within 3% of the actual values. The median error is 3%, and the largest error is 8%.

When we compared the measured *N_IPC* to the *N_IPC* computed using Eq.8, they were within 19% of each other on average (the median error was 10%). Although this produces the impression that the model is not sensitive to *N_IPC* estimates, this is not precisely the case. The model is more sensitive to *N_IPC* estimates for large N (e.g., N=3), because the probability that many threads are running is usually larger than the probability that few threads are running. Our estimates of *3_IPC* are actually much better than overall – within 5% of actual values on average, and with 10% being the largest error. When errors in *3_IPC* estimates are larger, the overall IPC estimates also suffer significantly.

This approach to modeling *N_IPC* is architecture dependent. Eq.8 will not work across different processor architectures, because the extent to which the IPC is affected when some hardware contexts are left unused greatly depends on how the processor schedules instructions, how many functional units it has, etc. Therefore, the relationship between *ideal_IPC* and *N_IPC* needs to be derived for a given microprocessor. Modeling this dependency precisely is a difficult problem.

4.2 Model evaluation

So far we have assumed that all threads running on a processor are executing identical workloads, and we have trained and tested our model using such workloads. It is not realistic to expect that real workloads would have such a property. Therefore, we now test our model using a heterogeneous workload.

To create heterogeneous workloads, we randomly combined the SPEC CPU integer benchmarks into groups of four. While we validated the model using a large number of such heterogeneous groups, in this paper we show validation experiments from only a sample of such groups, for clarity of presentation and in consideration for space. The sample that we chose is representative of the errors in estimated IPCs for all the groups we validated. We made sure that each SPEC benchmark is represented in at least one group in the sample. We present validation experiments from the following groups:

1. PVVV – 197.parser, 255.vortex, 175.vpr-place, 175.vpr-route
2. VVVC – 255.vortex, 175.vpr-place, 175.vpr-route, 186.crafty,
3. VVCG – 175.vpr-place, 175.vpr-route, 186.crafty, 176.gcc
4. VCGG – 175.vpr-route, 186.crafty, 176.gcc, 164.gzip
5. GMTV – 254.gap, 181.mcf, 300.twolf, 175.vpr-place
6. BEGV – 256.bzip2, 252.eon-cook, 176.gcc, 255.vortex

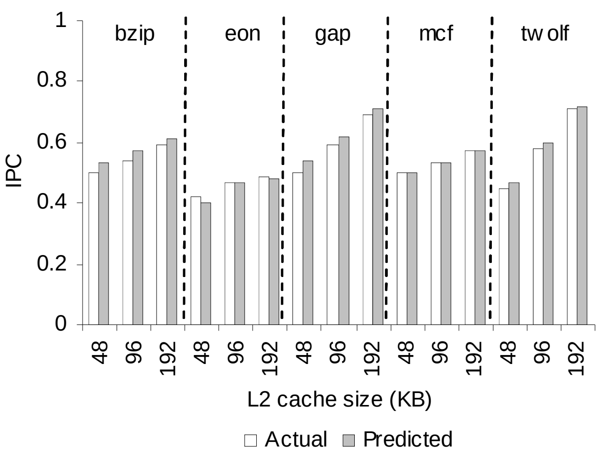

Figure 3. Actual vs. predicted IPC (modeled *N_IPC*, instead of measured *N_IPC*, substituted into Eq.7)

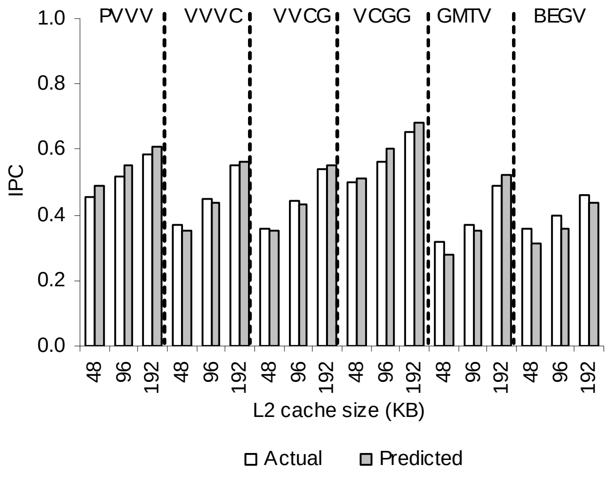

Figure 4. Actual vs. predicted IPC for a heterogeneous workload. Predicted IPC is on average within 6% of the actual.

We use the same methodology for obtaining the estimated *IPC_mt* and the actual IPC as described in Section 4.1.2.1, and we compute *N_IPC* using Eq.8As Figure 4 demonstrates, heterogeneity does affect our model to a certain extent: errors in IPC estimates for heterogeneous workload are 6% on average; this is 3% greater than for the homogeneous workload. The median error is 5%, and the largest is 14%.

Heterogeneity in the workload may violate the following two assumptions that we made. The first is the assumption that all threads have equal individual probabilities of being blocked. When several threads share the cache, some threads may have worse cache locality than others; those threads could contribute more to the overall miss rate, and have greater probabilities of being blocked. From the analysis of our data, we learned that when threads share the cache, they all experience the same individual miss rates. Therefore, the workload heterogeneity does not affect the assumption of equal blocked-probabilities.

The second assumption we made is that *N_IPC* for a given *N* is the same regardless of *which N* threads are running. However, when the workload is heterogeneous and all threads are different, the *N_IPC* for a given *N* does depend on which particular *N* threads are running, because each N-tuple of threads uses the processor resources in a unique way. For example, in benchmark group BEGV, the IPC when B, E, and G are running is not the same as the IPC when E, G, and V are running. Violation of this assumption is the cause for an increase in errors for heterogeneous workloads.

5. MODELING MEMORY-BUS DELAY

While the focus of our study was to model how multithreaded processors hide individual threads' memory delays, in order to evaluate our model on a realistic machine configuration, we had to factor in the delays that threads experience due to competition for the memory bus.

Memory can be thought of as a server responding to requests from clients. Clients are processors. In a multithreaded processor, clients are hardware contexts, because each hardware context issues memory requests independently. Although queuing theory is the canonical way to model such systems, we developed a simpler solution tailored to our specific problem. In Section 5.1, we explain how queuing theory could be used to model bandwidth delays and then in Section 5.2, we describe and evaluate our approach.

5.1. Using queuing theory

A canonical way to model contention for memory bus is using closed queuing network models with finite population [23, 24]. In this model, there is a finite population of customers that circulates within the system, and the arrival rate (and, consequently, the service time) depends on the number of customers that are already in the system. This model matches well to how the memory system operates: there are a fixed number of thread contexts, and the arrival rate of memory requests depends on how many threads are already waiting for the memory system. This arrival process is called state-dependent. Unfortunately, such queuing networks are difficult to solve – solutions usually do not have a closed form and involve iterative methods.

Matick et al have successfully used a simpler open-queue model to estimate memory-bus delays [18]. This model assumes that the arrival rate is independent of the state of the system. We attempted to use this model as well, and discovered that the independent arrival assumption is viable only if memory contention is low or moderate. When it is high, the model produced estimates for memory-bus delays that were sometimes off by as much as a factor of ten. Since for us it was important to accurately estimate both small and large delays, this method was not appropriate.

As an alternative, we developed a model that produces accurate estimates for a wide range of delays and has a closed-form solution.

5.2. Our approach

We represent our memory system as a server that answers requests for values stored in main memory. In our system, there are five concurrent streams of memory requests originating from four instruction streams and the write buffer (recall Section 3). We refer to these request originators as *threads*, and the number of request originators as *NUM_THREADS*.

Having sent a memory request, the thread spends some time waiting for the memory bus (*memory-bus delay*). Once the bus becomes available, the thread reserves it for a period of *WIRE_TIME* cycles (set to 80 in our simulator to correspond to the memory bandwidth of 1 GB/s) [3]. Once the request has been serviced, the thread goes away and computes for a while until it needs to send another request. We call the combination of *WIRE_TIME* and compute time the *request cycle window* (*REQUEST_CYCLE_WINDOW*) – this is the number of cycles that passes between the point in time when a request begins being serviced and the arrival of the next request. Figure 5 illustrates this.

The concept of *REQUEST_CYCLE_WINDOW* is at the heart of our memory-bus delay model, because the size of this window determines the intensity of competition for the memory bus. The *REQUEST_CYCLE_WINDOW expires* upon the arrival of a new request to the memory system. The shorter the window is, the more often it expires, the higher the request arrival rate, and the more intense the memory-bus competition. Since *WIRE_TIME* is fixed, the size of *REQUEST_CYCLE_WINDOW* is determined by the compute time – the time that the thread spends away from the memory bus. The greater this time the less the competition for the memory bus, and vice versa. We now show how to compute the size of *REQUEST_CYCLE_WINDOW*, and then explain how

[3] This memory bandwidth is more limited than that on modern processors. When bandwidth is less limited, the memory-bus delays are usually small. We wanted to test the ability of our model to estimate both small and large delays, so we configured the memory bandwidth to be more limited than in a realistic configuration.

to model the memory-bus delay using it.

REQUEST_CYCLE_WINDOW is equivalent to the number of cycles that pass between the arrivals of two subsequent memory requests from the same thread when there are no memory-bus delays (See Figure 5). We can compute this quantity using the IPC model for infinite memory bandwidth (Eq.7). Using the number of misses per instruction and the estimated *IPC_mt*, we can compute the number of misses per cycle. The inverse of misses-per-cycle is the number of cycles between two subsequent misses (i.e. arrivals) – this is the *REQUEST_CYCLE_WINDOW*.

We use the following equation to compute the number of misses per cycle (*L2_MR_CC*):

$$L2\_MR\_CC = \frac{(L2\_RMR + L2\_WMR) * IPC\_mt}{NUM\_THREADS} \qquad (9),$$

where *L2_RMR* and *L2_WMR* are the L2 read miss rate and L2 write miss rate (defined in Section 3) and *IPC_mt* is the IPC for the multithreaded processor (derived in equation 7). We divide the quantity in the numerator by the number of threads in order to compute the number of misses per cycle for a single thread; as we explained in Section 4.2, the assumption that all threads send memory requests at the same rate holds even for heterogeneous workloads.

We use the following equation to compute the size of *REQUEST_CYCLE_WINDOW*:

$$REQUEST\_CYCLE\_WINDOW = 1 / L2\_MR\_CC \qquad (10).$$

Now, let us show how to model the memory bus delay using *REQUEST_CYCLE_WINDOW*. We model the memory-bus delay from the point of view of an individual thread, t0, and we assume that all threads experience the same delays.

We assume that if all other threads send their requests at the same non-bursty rate as t0, those threads' requests will arrive to the memory system sometime during t0's *REQUEST_CYCLE_WINDOW*. As a result, the amount of time that t0 has to wait for the memory bus once its *REQUEST_CYCLE_WINDOW* expires depends on a) the size of *REQUEST_CYCLE_WINDOW* and b) at which position in this window the other threads' requests arrive. Intuitively, if the other threads' requests arrive early in t0's window, they may have time to finish before t0's window expires (if the window is large enough). T0, then, will not have to wait for the memory bus when it sends its next request. If the other requests arrive late, on the other hand, then they will not finish before t0's window expires, and t0 will need to wait.

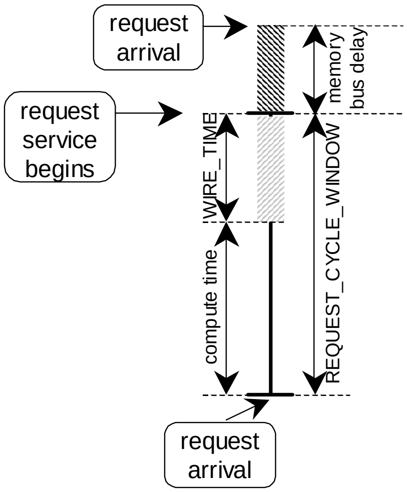

Figure 5. The request cycle window.

We divide the *REQUEST_CYCLE_WINDOW* into two portions: the *top portion* and the *bottom portion,* where the top portion corresponds to the early arrival of the other threads' requests, and the bottom portion corresponds to the late arrival.

We model the memory-bus delay by estimating the wait times for the top and bottom portions, computing the probabilities that the requests' arrival falls into a particular portion, and then weighing the delay in each portion by its respective probability. Figure 6 illustrates how we divide the window into the portions and how we estimate the delays for each portion. We now describe this in detail.

The top portion starts at the top of the window. If the window is large enough such that the other threads' requests can be serviced before *REQUEST_CYCLE_WINDOW* expires, the top portion stretches until the latest point at which those requests must arrive so that they can free the wire before the window expires. Figure 6a) illustrates this case. If the other threads' requests arrive in the interval of time covered by the top portion of the window, t0's associated wait time will be zero.

However, if the window is not large enough, then the top portion covers the stretch of the window when the wire is still reserved by t0's memory request – this is the top interval of the window equal to *WIRE_TIME* (Figure 6b). In this case, t0 will need to wait for the duration of time equal to *(WIRE_TIME * NUM_THREADS - REQUEST_CYCLE_WINDOW)* once its next request arrives. Figure 6b) illustrates this.

We compute the wait times associated with the top portion of the window using the following formula:

$$\begin{aligned} WAIT\_TOP = \quad & MAX(0, \\ & WIRE\_TIME * NUM\_THREADS - \\ & REQUEST\_CYCLE\_WINDOW) \qquad (11). \end{aligned}$$

The bottom portion of the window is the part of the window not covered by the top portion. The minimum amount of wait for the bottom portion equals to *WAIT_TOP* (Eq.11), because the bottom portion commences where the top portion ends. To understand what the maximum wait would be, consider Figure 6c. The maximum wait for the bottom portion occurs if the other threads' requests arrive just before the *REQUEST_CYCLE_WINDOW* expires. The wait in this case is equal to the amount of time it takes to service those threads' requests, *WIRE_TIME * (NUM_THREADS – 1)*. We summarize the minimum and the maximum wait times associated with the bottom portion below:

$$MIN\_WAIT\_BOTTOM = WAIT\_TOP$$

$$MAX\_WAIT\_BOTTOM = WIRE\_TIME * (NUM\_THREADS - 1).$$

The key distinction between the top and the bottom portions is that the wait time in the top portion is fixed (the exact wait time depends in the window size). The wait time in the bottom portion ranges between *MIN_WAIT_BOTTOM* and *MAX_WAIT_BOTTOM.* To understand this, consider Figure 6d. We compute the wait time corresponding to the bottom portion as the average of the minimum and maximum wait times:

$$WAIT\_BOTTOM = \frac{MIN\_WAIT\_BOTTOM + MAX\_WAIT\_BOTTOM}{2} \tag{12}$$

From our definitions of the top and bottom portions, the lengths of these portions can be easily computed:

$$\begin{aligned}&top\_portion = \\ &MAX(WIRE\_TIME; \\ &REQUEST\_CYCLE\_WINDOW - NUM\_THREADS * WIRE\_TIME)\end{aligned}$$

$$bottom\_portion = REQUEST\_CYCLE\_WINDOW - top\_portion$$

And the probability of the requests arriving in a given portion is simply the fraction of the *REQUEST_CYCLE_WINDOW* that the portion occupies:

$$P(arrive\_at\_top) = \frac{top\_portion}{REQUEST\_CYCLE\_WINDOW}$$

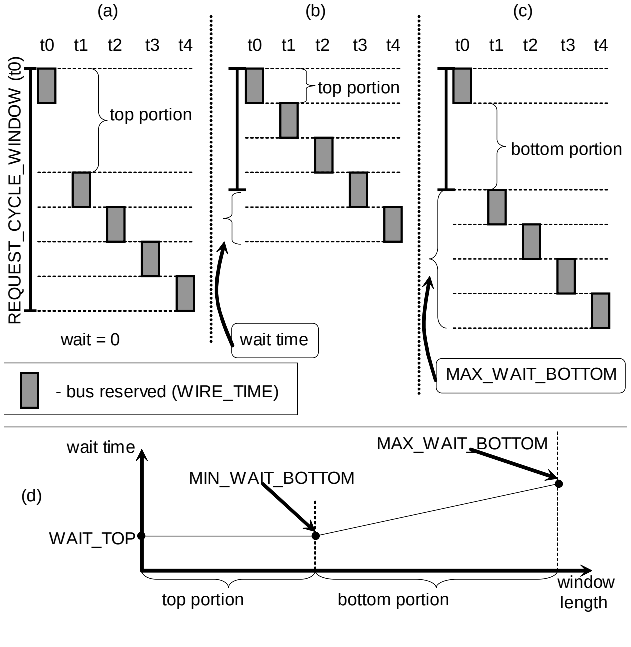

Figure 6. Shows how the memory-bus wait time for t0 depends on how requests of the other four threads are positioned in its *REQUEST_CYCLE_WINDOW.*

$$P(arrive\_at\_bottom) = \frac{bottom\_portion}{REQUEST\_CYCLE\_WINDOW}$$

To compute the overall memory-bus delay per transaction (*MEM_BUS_DELAY*), we weight the wait times associated with the top and bottom portions by their respective probabilities:

$$\begin{aligned}MEM\_BUS\_DELAY = \\ &P(arrive\_at\_top) * WAIT\_TOP + \\ &P(arrive\_at\_bottom) * WAIT\_BOTTOM \qquad (13).\end{aligned}$$

The quantity in Eq. 13 expresses the expected memory-bus delay per L2 miss transaction. In the next section we evaluate its accuracy.

5.3. Evaluation and discussion

We used the heterogeneous workload described in Section 4.2 to validate how well we can estimate memory-bus delays. Using Equation 13 we were able to estimate memory-bus delays to within 12% of the actual quantities, on average (See Figure 7). The median error was 10%, and the largest was 25%.

We made a simplifying assumption that all threads send memory requests at the same rate. While this holds for the four hardware contexts, as we explained in Section 4.2, the requests from the write buffer usually arrive at a different rate. While we observed that for our workload this dissimilarity did not affect the accuracy of memory-bus delay estimates, we believe that this phenomenon should be investigated further, for a wider range of workloads.

Another assumption that we made when describing how requests of the other threads arrive from the viewpoint of a particular thread is that those requests arrive in lockstep. We believe that this assumption does not hurt our estimates. By representing the hypothetical positioning of other threads' request arrivals we tried to capture both the best and the worst cases. Lockstep arrival positively affects the wait time for the best case

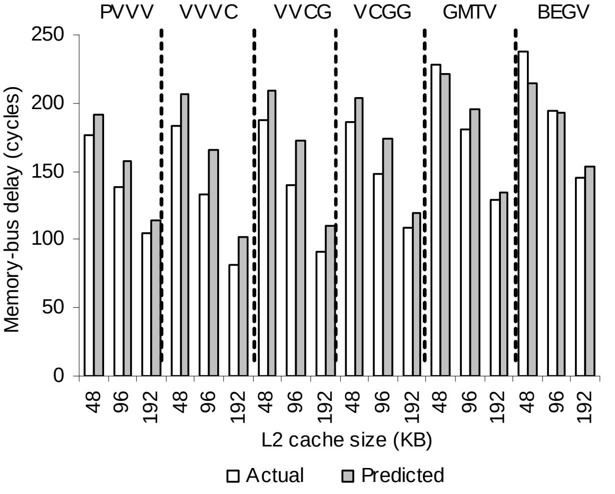

Figure 7. Actual vs. estimated memory-bus delay per transaction.

and exacerbates the wait time for the worst case. Therefore, this assumption, in fact, helps us to define the two extremes.

Although our model for memory-bus delays is simple, it works, because the basis for our approach is well-matched to the problem: we identify the best and the worst wait times that a thread may experience and assign rough probabilities to these wait times. We base those probabilities on the size of the *REQUEST_CYCLE_WINDOW*, which, as we explained in Section 5.4 determines the magnitude of contention. Based on this observation and the validation results, we believe that our model is robust against a wide range of workloads and input parameters. We also believe that this model can be used for other multithreaded architectures and in general for multiprocessor systems, because the underlying causes of memory contention are the same irrespective of processor architecture.

6. PUTTING IT ALL TOGETHER

In Section 4 we described how we model the latency-hiding effect of hardware multithreading. In Section 5 we presented a model for memory-bus delay. Now we put these two pieces together and validate the entire model.

Eq.13 gives us the memory-bus delay per L2 miss transaction. To factor this delay into our IPC model (Eq.7), we augment the cost of handling the L2 miss, *L2_MCOST*, by *MEM_BUS_DELAY*. This changes Eq.4 as follows:

$$L2\_CPI = (L2\_RMR + L2\_WMR * WMM) * \\ (L2\_MCOST + MEM\_BUS\_DELAY) \qquad (14).$$

We use this version of the equation to compute *L2_CPI*, which we substitute into Eq.6, and eventually into Eq.7 to estimate *IPC_mt*.

To validate the accuracy of the estimated *IPC_mt* we use the same group of heterogeneous workloads described in Section 4.2. We run these benchmarks on a simulated system configured with memory bandwidth limited to 1 GB/s and measure the resulting IPC. Figure 8 shows how it compares to *IPC_mt*.

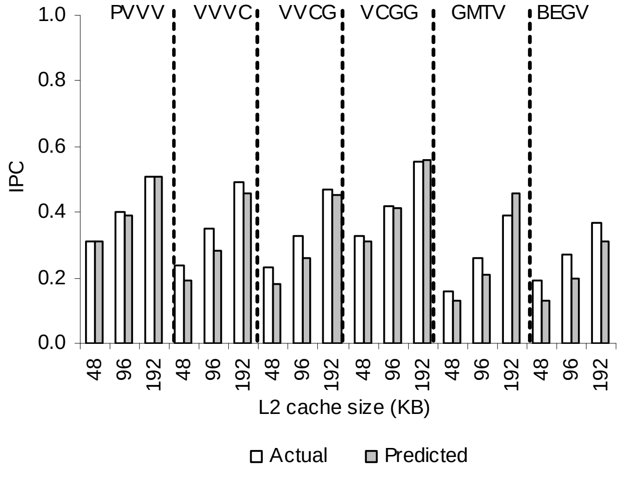

Figure 8. Actual vs. estimated IPC with limited memory bandwidth.

On average, the estimated *IPC_mt* is within 13% of the actual. The median error is 16%, and the largest error is 32%. In all but one case, the direction of the error for a given workload is consistent across cache sizes: we either consistently overestimate or underestimate the IPC; the magnitude of error is usually consistent for a given workload. This means that our model is successful at predicting the general performance trend and the magnitude of the effect of cache miss rate on IPC even when the estimates are not precise.

Our model is successful across a wide range of input parameters: it has worked for L2 miss ratios ranging from as low as 1% to as high as 46%, and has predicted memory-bus delays from as low as 82 cycles per transaction to as high as 238. The workload contained benchmarks with diverse instruction mixes and localities of reference, which is demonstrated by the range of *ideal_IPC_mt*: from a high of 0.96 (multithreaded bzip) to a low of 0.49 (multithreaded eon). The reason for the effectiveness of our model is tied to the abstractions that we used to model latency hiding and memory-bus delays. We developed a technique to model latency-hiding in a way that is not dependent on the cache architecture or the workload. For memory-bus delay, we observed that the magnitude of contention is determined by how much time each thread spends away from the memory bus, and used the abstraction of *REQUEST_CYCLE_WINDOW* to represent the degree of contention.

Because our model is based on the principles that are not tied to details of processor microarchitecture, we are confident that it will work for a variety of multithreaded processors. Verifying this is an area of future research.

7. USING THE MODEL

In previous work we proposed an operating system scheduler tailored for multithreaded chip multiprocessors (CMT) – processors that have multiple multithreaded processor cores on a single chip [20]. We found that such a scheduler has the potential to reduce L2 miss ratios by 19-37% and improve processor IPC by 27-45%. We are now working on implementing this scheduling algorithm, and we plan to use our IPC model in the scheduler.

The scheduler reduces L2 contention by employing a balance-set approach: it arranges threads in *schedules*, such that threads in the schedule produce low L2 miss rate when run together. The scheduler then assigns each schedule to run for a time slice.

One challenge in implementing this algorithm is deciding how many threads should belong to a schedule. Ideally, it should be the same as the number of the available hardware contexts, because leaving contexts unused may poorly affect performance. However, scheduling too many threads may cause thrashing in the L2, and negatively affect processor IPC. There is a clear trade-off.

Our scheduler needs to decide (on-line) whether it pays to trade unused hardware contexts for better performance in the L2 for a given workload. Our IPC model would aid it in making the decision. The scheduler would compute the L2 miss rates (as described in the scheduling paper [20]) for various schedule sizes. Then, using our model, it would estimate the corresponding IPCs. It would choose the schedule size with the best estimated IPC.

We also plan to use our model to address fairness in the scheduler. The nature of the balance-set scheduling algorithm creates pressure to schedule cache-frugal threads more often than cache-greedy threads, trading-off fairness for performance. Our IPC model can serve to estimate the magnitude of performance gains for thread schedules with different levels of fairness, and help the scheduler decide how much fairness to give up.

8. RELATED WORK

The techniques we used for modeling IPC for single-threaded workloads were inspired by Denning's work on modeling the relationship of processor utilization and the performance of the virtual memory system [19]. Denning's study offers some basic approaches, such as using ideal IPC in the model. His model is high-level, it does not take into account details of the hardware and is not validated against real data.

Matick et al. describe a model for multi-processor IPC based on cache miss rates [18]. Their model of cache delays for single-threaded workloads uses approaches similar to ours. The fundamental distinction of our work is that we model how a multithreaded processor hides individual threads' memory latencies – this has not been addressed in Matick's study or elsewhere.

9. CONCLUSIONS

We presented a model for the relationship between the performance of on-chip caches and processor IPC for multithreaded processors. The fundamental challenge that we addressed in our work was to model how a multithreaded processor hides cache-related delays experienced by the threads.

We also presented a model for memory-bandwidth delays that uses a simple approach, but produces accurate estimates.

Our model has a closed-form solution and produces IPC estimates that are on average within 13% of actual quantities. Because the model is based on simple principles that do not depend on processor architecture, it works for a wide range of workloads and input parameters. We are optimistic that for this reason it will work for a variety of multithreaded processors, irrespective of the details of their microarchitecture.

By developing this model we have paved the way towards a better understanding of how the performance of multithreaded processors depends on the interactions between its components. Such an understanding will help designers make beneficial tradeoffs both in multithreaded hardware architectures and in the software that runs on these new systems.

10. REFERENCES

[1] J. Lo et al., "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors", *ISCA'98*.

[2] A. Ailamaki, D. DeWitt, M. Hill, D. Wood. "DBMSs on modern processors: Where does time go?" *VLDB `99.*

[3] A. Barroso, K. Gharachorloo, E. Bugnion, "Memory System Characterization of Commercial Workloads", *ISCA'98.*

[4] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads", *ISCA'98.*

[5] D. Tullsen, S. Eggers, H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *ISCA'95.*

[6] R. Alverson et al., "The Tera Computer System", *Proc. 1990 Intl. Conf. on Supercomputing*.

[7] A. Agarwal, B-H. Lim, D. Kranz, J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing", *ISCA'90*

[8] J. Laudon, A. Gupta, M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations", *ASPLOS VI*, October 1994.

[9] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, D. Tullsen, "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading", *ACM TOCS 15, 2,* August 1997.

[10] D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology Journal Q1,* 2002.

[11] L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", *ISCA'00*.

[12] D. Nussbaum, A. Fedorova, C. Small, "The Sam CMT Simulator Kit.", *Sun Microsystems TR 2004-133*, 2004.

[17] SPEC CPU2000 Web site: http://www.spec.org/cpu2000/analysis/memory/

[18] R. E. Matick, T. J. Heller, and M. Ignatowski, "Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory", *IBM Journal Of Research And Development, Vol. 45 NO. 6,* November 2001.

[19] P. Denning, "Thrashing: Its causes and prevention", *Proc. AFIPS 1968 Fall Joint Computer Conference*, *33, pp. 915-922*, 1968.

[20] A.Fedorova et al., "Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design", *Sun Microsystems TR 2004-0797*, 2004

[21] A. Barroso, K. Gharachorloo, E. Bugnion, "Memory System Characterization of Commercial Workloads", *ISCA'98.*

[22] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads", *ISCA'98.*

[23] R. Onvural, "Survey of Closed Queuing Networks With Blocking", *ACM Computing Surveys, v. 22, issue 2, pp. 83-121*, 1990

[24] L. Kleinrock, "Queuing Systems Vol I", *Wiley*, 1975.

[25] "IBM eServer iSeries Announcement", http://www-1.ibm.com/servers/eserver/iseries/announce/

[26] Jonathan Schwartz on Sun's Niagara processor: http://blogs.sun.com/roller/page/jonathan/20040910#the_difference_between_humans_and

[27] Intel web site, http://www.intel.com/pressroom/archive/speeches/otellini20030916.htm

[29] J. M. Borkenhagen et al., "A multithreaded PowerPC processor for commercial servers", *IBM Journal Of Research And Development, Vol. 44 NO. 6,* 2000.

[28] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, David J. Lilja, and Vijay S. Pai, "Challenges in Computer Architecture Evaluation", *IEEE Computer,* August 2003.