

Performance Tradeoffs in Multithreaded Processors

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

High network latencies in large-scale multiprocessors can cause a significant drop in processor utilization. By maintaining multiple process contexts in hardware and switching among them in a few cycles, multithreaded processors can overlap computation with memory accesses and reduce processor idle time. This paper presents an analytical performance model for multithreaded processors that includes cache interference, network contention, context-switching overhead, and data-sharing effects. The model is validated through our own simulations and by comparison with previously published simulation results. Our results indicate that processors can substantially benefit from multithreading, even in systems with small caches, provided sufficient network bandwidth exists. Caches that are much larger than the working-set sizes of individual processes yield close to full processor utilization with as few as two to four contexts. Smaller caches require more contexts to keep the processor busy, while caches that are comparable in size to the working-sets of individual processes cannot achieve a high utilization regardless of the number of contexts. Increased network contention due to multithreading has a major effect on performance. The available network bandwidth and the context-switching overhead limits the best possible utilization.

1 Introduction

As we build larger and larger parallel machines, the proportion of processor time actually spent in useful work keeps diminishing. There are several reasons for the decreasing processor utilization. First, the cost of each memory access increases because network delays increase with system size. Higher processor clock rates will only magnify this effect. Second, as we strive for greater speed-ups in applications through fine-grain parallelism, the number of network transactions and synchronization delays also increases.

A method for improving processor utilization is to multithread the processor. Such processors switch to a new thread and perform useful computation while other threads wait for memory responses or synchronization signals. While multithreading has usually meant cycle-by-cycle interleaving of instructions from different processes, we apply the same term to systems that interleave blocks of instructions from different processes as well. The cycle-by-cycle interleaved processors are called *finely* multithreaded processors, and the others are called *coarsely* multithreaded processors or *block* multithreaded processors. Several processor designs have used multithreading to mask communication and synchronization latencies, or to utilize deep pipelines effectively, e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. By multithreading a processor such that an instruction from a different thread can be initiated every cycle (or every few cycles), pipeline bubbles due to pipeline dependencies or processor stalls due to memory latency can be prevented. Processors in

message-passing multicomputers often maintain multiple processes per node and context switch among them to overlap message latencies [11, 12, 13].

There are limits to the improvements in processor utilization achievable by multithreading a processor. Most important, multithreading requires applications to display sufficient parallelism to permit multiple threads to be assigned to each processor. (This paper uses the terms process, thread, task, and context interchangeably.) Provided sufficient parallelism exists, the improved processor utilization must be traded off against negative cache and network effects. In machines with caches, multiple simultaneously active processes interfere with each other in the cache and give rise to a higher cache miss rate and hence a higher network traffic rate. Similarly, higher utilizations in multithreaded processors increase the demand on network bandwidth. Finally, the processor might waste a few cycles while switching between processes. These cycles constitute the context-switching overhead. Thus, a multithreaded processor design must address the tradeoff between higher utilization and increased cache miss rates, context-switching overhead, and network contention.

Our analysis is aimed at quantitatively understanding the performance tradeoffs involved in designing a multithreaded processor. What are the limits to the improvement in processor utilization as we increase the number of processes? How do cache and network design impact these limits? What is a reasonable context-switching overhead? To answer these questions, we derive a model of multithreaded processor performance that takes into account deleterious cache and network effects. The model predicts performance as a function of the number of processes among which the processor can rapidly switch. We also refer to this number of processes as the number of processor-resident threads, or the number of active threads per processor. This model provides insights into the relationships between the various factors involved and estimates of expected processor utilization. Thus, we can use this model to indicate the domains of feasibility for multithreaded processors.

As an indication of our results, for a specific set of parameters, when the number of processor-resident threads increases from one to two, processor utilization increases from 0.4 to 0.7. However, due to increased cache interference and context-switching overhead, processor utilization only increases from 0.7 to 0.8 when the number of processor-resident threads goes from two to three. We also show that high processor efficiencies can be achieved in block multithreaded systems with context-switching overheads in the 10-cycle range.

Clearly, our performance predictions require a caveat. The model neglects several important concerns, such as the availability of enough parallel threads, impact of register file size on the clock cycle, and context-switch decision making. Furthermore, our processor-utilization measure does not account for processor cycles wasted in thread management. Because these parameters are closely related to the characteristics of parallel applications and implementation constraints, the ultimate test of the performance of multithreaded processors can only be made in an actual system implementation. The April processor architecture and the software system design of the Alewife multiprocessor (described in the next section) is aimed at investigating these issues in more detail. However, the model presented in this paper can still be used to evaluate tradeoffs in the design process.

The rest of the paper is organized as follows. Section 2 discusses practical design issues and reviews our ongoing multithreaded processor design effort. Section 3 presents the multithreaded processor performance model. This model includes the network model described in Section 4 and the multithreaded cache model of Section 5. The multithreaded processor model is summarized and validated in Section 6. Section 7 uses this model to analyze the tradeoffs in multithreaded

processors. Section 8 compares our results to previous analyses of multithreaded processors and presents more validations. Section 9 presents directions for future work and the current status of our project. Section 10 concludes the paper.

2 Designing a Multithreaded Processor

We have designed a multithreaded processor architecture called April as part of the Alewife multiprocessor project at MIT [10]. Alewife is a large-scale, cache-coherent machine, with globally-shared memory that is physically distributed among the processing nodes. Cache coherence is maintained using a distributed directory scheme [14], and remote memory transactions are satisfied over a mesh network. The Alewife project focuses on techniques for automatic latency minimization and automatic latency tolerance in large-scale multiprocessors. The compiler, runtime system, and caches cooperate in minimizing the latency of memory operations by trying to enhance locality through partitioning and dynamic migration of processes and data. When a memory request is forced to traverse the interconnection network, the processor tolerates its latency by rapidly switching to another process. The cache controller in each Alewife node can activate either the trap line or the wait line to the processor. The trap line forces a context switch during a remote transaction or on an unsuccessful synchronization test, while the wait line busy-waits the processor during short transactions such as cache misses directed to local memory.

A desirable multithreaded processor architecture has several important properties, two of which are a low context-switching overhead and a high single-thread performance. As shown in Section 7, switching overhead limits the maximum attainable processor utilization and establishes the minimum latency that can be tolerated profitably. In particular, if we wish to tolerate pipeline latencies, the overhead must be smaller than the pipeline depth. High single-thread performance is important to run sections of applications with low parallelism efficiently.

Unfortunately, the above goals are hard to achieve simultaneously. High single-thread performance requires maximizing the amount of processor-resident state, while a fast context switch has the opposite demand. Achieving high single-thread performance requires allowing multiple instructions from the same process in consecutive stages of the processor pipeline. This increases the amount of state that must be saved and restored on a context switch, while also increasing the difficulty of cleanly halting the pipeline. A switch to a new thread must save the program counters and the processor status word, increment the context pointer, and restart the pipeline. Program counters and status words can be saved in register frames, or they can be implemented in separate per-processor frames (like registers) to reduce the switching time. In contrast, the effect of pipeline flushing is harder to minimize without an accompanying mechanism to speedily save and restore the entire pipeline state.

The opposing goals of high single-thread performance and fast context switches have been previously addressed largely in their extremes. Finely multithreaded processors [2, 3, 5, 7] that disallow the execution of consecutive instructions from the same process can support very fast context switches because the various instructions in the pipeline at any given time are independent. Consequently, they can use multithreading to utilize deep pipelines efficiently in addition to hiding network latencies. However, they suffer from poor single-thread performance. Most other processor designs achieve high single-thread performance using compilers to enhance pipeline performance [15], but cannot switch contexts rapidly.

In contrast, April achieves high single-thread performance by using pipeline bypass paths

and compiler pipeline optimization, and provides hardware support to save and restore process state efficiently. The hardware design is simplified by the fact that processors in cache-coherent machines can withstand modest context-switch overheads (in the range of 4 to 12 cycles). As discussed in Section 7.3, the expected lower frequency of remote memory operations in a cache-coherent multiprocessor mitigates the negative effects of higher context-switch overheads.

April obviates register saves and restores across context switches by using a register file organized into several register frames, each of which implements a hardware context that can store the state of a process. Each process uses two register frames. The first frame is used for process registers, the second for registers required by the trap handler. A context switch to a process whose state is currently stored in one of the register frames on the processor is effected in a small number of cycles.

The Alewife machine supports unlimited virtual processes. The mapping of process contexts to register frames is managed by software. In other words, processor register frames act as a software-controlled cache of process contexts.

The current implementation of April using a SPARC processor [16] modified to support block multithreading is called Sparcle. Sparcle was implemented jointly with LSI Logic and SUN Microsystems; a single-node Sparcle system has been operational since March 1992. The register set in Sparcle is divided into several frames that are conventionally used as register windows [17, 18] for speeding up procedure calls. SPARC permits the use of these frames for context switching because the frame pointer is incremented in software by a special instruction that is not strictly tied to the procedure call. In our design, a process does not use multiple register windows. Instead, we partition the register file into four hardware contexts. Several studies have shown that single process frames, combined with register allocation methods, can achieve comparable performance to register windows (e.g., see [19, 20]). Our hardware modifications improve SPARC's switching efficiency and allow multiple-context partitioning of the registers in the floating-point coprocessor as well. See [10] for more details.

A processor that permits rapid context switching and fast trap handling, facilitated by the same mechanisms as fast context switches, has added advantages. In Alewife, fast trap handling allows efficient migration of some cache coherence functionality into the software runtime system [14]. The controller design is simplified because it relegates the responsibility of handling exceptional situations, such as special message arrival or network buffer overflow, to the processor. Rapid context switching makes the use of coherence protocols that guarantee sequential consistency feasible because the processor can switch to a different thread while awaiting acknowledgments to outstanding memory transactions. Rapid trap handling also allows efficient handling of synchronization faults.

3 A Performance Model for Multithreaded Processors

A model for a multithreaded processor must represent the tradeoff between increased processor utilization due to overlapping network access with useful computation and the higher cache miss rates and network contention. For the purpose of this analysis, we assume that the processes resident on a processor have the same cache miss rates and that between misses the processes execute useful instructions. Context switches happen only on cache misses, and processor cycles spent in context switching are considered wasted.

If p is the number of processes resident on a processor (or the number of hardware contexts),

Figure 1: Hiding network latency by multithreading the processor.

let the time between misses for each process be $t(p)$, the time to satisfy a miss be $T(p)$, and the time wasted in context switching be C . (Section 3.1 also considers exponentially distributed inter-cache-miss times and cache miss service times.) A process executes useful instructions for $t(p)$ cycles, suffers a cache miss, and then waits $T(p)$ cycles for the miss request to be satisfied before it can proceed. Context switches happen only on cache misses. Consequently, $t(p)$ represents the *context-switching interval*. Time wasted in context switching is called the *context-switching overhead*. We measure time in terms of processor cycles.

As depicted in Figure 1, some of the time expended to satisfy a cache miss can be overlapped with useful processor execution. With p available processes and no context-switching overhead, effective processor utilization is

$$U(p) = \frac{p t(p)}{t(p) + T(p)}$$

with a maximum utilization of 1.

Context-switch overhead can be factored in easily. We assume that the cache miss rate is independent of the context-switch overhead and that the instructions executed during a switch do not cause any cache misses. This is a reasonable assumption because the code executed during the context switch is either cache resident due to frequent use or hardwired. The context-switch overhead is independent of the number of hardware contexts in our model. If the context-switch overhead is C processor cycles, then the utilization equation remains the same for all p such that

$$p [t(p) + C] < t(p) + T(p)$$

because the number of useful cycles during the interval $t(p) + T(p)$ is still $pt(p)$. When the above inequality is not satisfied, the utilization becomes limited by the context-switch overhead. For every $t(p)$ useful cycles, the processor wastes C context-switch related cycles, yielding a limiting utilization of

$$U(p) = \frac{t(p)}{t(p) + C} \quad \text{for } p \geq \frac{t(p) + T(p)}{t(p) + C}$$

If $m(p)$ is the cache miss rate, defined as the probability of a miss on a non-idle processor cycle, and a context switch is forced on each miss, then $t(p)$ is simply the inverse of the miss rate. In practice we might force a context switch only on a miss to a nonlocal memory module. Since $t(p) = 1/m(p)$, the utilization becomes¹

¹If the context-switch overhead is considered part of $t(p)$ and if it contributes to the miss rate, the processor utilization for $p < 1 + T(p)m(p)$ is given by

$$U(p) = \frac{p(1 - Cm(p))}{1 + T(p)m(p)}$$

$$U(p) = \begin{cases} \frac{p}{1+T(p)m(p)} & \text{for } p < \frac{1+T(p)m(p)}{1+Cm(p)} \\ \frac{1}{1+Cm(p)} & \text{for } p \geq \frac{1+T(p)m(p)}{1+Cm(p)} \end{cases} \quad (1)$$

Now we need to express both $m(p)$ and $T(p)$ in terms of the number of processor-resident threads p . We start by summarizing the terms used thus far in Table 1, and the assumptions made in the rest of our analysis.

p	Degree of processor multithreading
$T(p)$	Number of processor cycles to satisfy a cache miss
$t(p)$	Number of processor cycles between misses, or the inter-cache-miss time
$m(p)$	Cache miss rate
$U(p)$	Processor utilization
C	Context-switching overhead in processor cycles

Table 1: Definition of terms used in the equation for processor utilization.

3.1 Assumptions

Our analysis includes the following general assumptions and simplifications.

- All processes resident on a processor have the same cache miss rate and working-set size.
- Network requests and context switches occur only on cache misses and coherence-related invalidations. The multithreaded cache model computes the cache miss rate as a function of p , the degree of processor multithreading. The invalidation component depends on application characteristics and the size of the multiprocessor. Its effect is incorporated in our analysis by adding a constant m_{inv} to the miss rate.
- Our analysis does not address the impact of multithreading on synchronization latencies. However, our analysis can be extended easily to do so by adding the product of the rate of synchronization faults and the average synchronization delay to the $1 + T(p)m(p)$ terms in Equation 1. The ability to overlap these synchronization delays will make multithreading appear relatively more useful.
- Initially, we do not consider the effect of processes sharing data in the cache. While low levels of data sharing occur in many applications, sharing of instruction blocks and read-only data is expected (and must be encouraged). Furthermore, affinity scheduling disciplines that favor same-processor execution of threads operating on overlapping data sets will also increase the shared fraction in the cache. We are actively investigating several such methods. In general, data sharing does not change the form of our results because its overall effect is to reduce the effective size of the process working sets. Our results allow for such variations by considering a range of working-set sizes. For completeness, the miss-rate model is extended in [21] to account for sharing effects by reducing the size of the individual process working sets by the shared fraction.

and for $p \geq 1 + T(p)m(p)$ is given by

$$U(p) = 1 - Cm(p)$$

- Our results do not include the effect that the nonstationary behavior of the process has on the cache. A process suffers nonstationary misses when it must renew parts of its working set. When a process returns to a processor, some fraction of its working set is renewed. If we do not account for this effect, the analysis will incorrectly associate some cache misses with multithreading rather than with the nonstationary behavior of the program. Even though we have found that this effect is not significant for most applications, we modify the model to account for this effect in [21]. It is interesting to note that applications that suffer from a high nonstationary miss-rate component will have a relatively low multithreading-related miss-rate component. Errors in estimating the latter component will not significantly impact the results.
- Our processor-utilization model assumes a fixed time interval between context switches and a fixed cache miss service time. Alternately, we could assume a fixed probability of a cache miss on any cycle of processor execution, leading to geometrically distributed time intervals between context switches with mean $t(p)$ and exponentially distributed cache miss service times with mean $T(p)$. For these alternate distributions, the processor utilization can be derived from a simple M/M/1/M queueing model [22] for a finite population of size p , a single queueing server (the processor) with service times that are exponentially distributed, whose mean is $t(p)$, and the network modeled as a delay center with service times that are exponentially distributed, and whose mean is $T(p)$. In this case, the processor utilization is one minus the probability that the queueing server is idle, or

$$U(p) = 1 - \frac{1}{\sum_{q=0}^{q=p} \left(\frac{t(p)}{T(p)}\right)^q \frac{p!}{(p-q)!}} \quad (2)$$

Simplifying and ignoring second and higher order terms in the above summation we get $U(p) = p/(p + T(p)/t(p))$, which is similar in form to the expression obtained with fixed values for $t(p)$ and $T(p)$. The added p term in the denominator lowers the computed value of the utilization.

We will make additional assumptions that have been traditionally made in network and cache analysis, and we will point these out when we do so. We continue by describing the multithreaded network and cache models in Sections 4 and 5 respectively, and then summarizing the chief results of these models in Section 6.

4 Modeling the Effect of Network Latency

This section derives the cache miss service time $T(p)$ as a function of the degree of multithreading. $T(p)$ includes the network latency and memory access times. We use a packet-switched, direct interconnection network [23] of the k -ary n -cube class. The number of processors in a k -ary n -cube network is k^n . As an example, an 8-ary 2-cube is a two-dimensional mesh with 64 processors. The network uses cut-through routing [24], and messages are routed completely in one dimension before the next. We will suggest the changes to the model if an indirect network [25] is used instead. Our results show that the indirect network is less sensitive to the increased bandwidth requirements of multithreaded processors than the direct network, although the indirect network is expected to be costlier.

Our network analysis assumes:

- Uniform traffic rates from all the nodes.
- Infinite buffering at the switch nodes. Simulation experiments [26] have shown that as few as four packet buffers at each switch node can approach infinite buffer performance.
- Uniformly distributed and independent message destinations.

The network parameters are summarized in Table 2. The message size is B times the network channel width. For notational convenience, in the remainder of this paper we drop the dependence of t and T on the number of processes.

M	Memory access time
B	Message size
n	Network dimension
k	Network radix

Table 2: Definition of terms used in the network model.

A simple performance model for buffered, k -ary n -cube, direct networks is derived and validated in [27]. From the model, the average cache miss service time T is given by,

$$T = \left(1 + \frac{\rho B \frac{1}{k_d} \left(1 - \frac{1}{k_d} \right)}{(1 - \rho)} \right) h + M + B - 1 \quad (3)$$

where ρ is the network channel utilization, B is the message size, and k_d is the average distance a message travels in any dimension. The delay T is computed as the sum of the memory access time (M), the pipeline delay of the message ($B - 1$), and h network switch delays.² The h hops through the network correspond to $h/2$ each for the request and the response or acknowledgment. The delay at each switch stage is one plus the queuing delay. (As shown in [27], separate treatment of the incoming and outgoing messages from the processing nodes adds an additional $(1 + 1/n)$ factor to the queuing delay, and results in greater accuracy.)

Assuming separate channels for both directions, the average distance traveled in a dimension is computed as the expected number of hops between two randomly chosen nodes in a one-dimensional array of nodes. The expected number of hops in a one-dimensional array can be determined by summing the distances for all unique source-destination pairs, and dividing by the total number of such pairs. Thus

$$k_d = \frac{k - \frac{1}{k}}{3} \approx \frac{k}{3}$$

We can determine ρ as follows. The probability of a network request on any given cycle from a processing node is $2p/(t + T)$ (see Figure 1). The factor of two accounts for both the messages generated by memory requests and responses. On average, a message of size B travels k_d hops in each of n dimensions, for a total of $h/2 = nk_d$ hops. Because each switch has $2n$

²For indirect networks, we can use the model presented in [26], where

$$T = \left[1 + \frac{\rho B \left(1 - \frac{1}{k} \right)}{2(1 - \rho)} \right] h + M + B - 1$$

associated channels (separate channels for each direction exist), the channel capacity consumed, or the channel utilization, is given by,

$$\rho = \frac{2p}{t+T} \frac{Bnk_d}{2n} = \frac{p}{t+T} Bk_d \quad (4)$$

Let the network delay without contention be denoted $T_0 = h + M + B - 1$. Substituting the expression for ρ from Equation 4 into Equation 3, and setting $t = 1/m$ and $k_d = k/3$, we solve for T as a function of p as shown below.

$$T = \frac{T_0}{2} + \frac{Bpk}{6} - \frac{1}{2m} + \frac{1}{2} \sqrt{\left(T_0 - \frac{Bpk}{3} + \frac{1}{m}\right)^2 + 8pB^2n\frac{k}{3} \left(1 - \frac{3}{k}\right)} \quad (5)$$

The above expression implies that T increases roughly as the number of threads p ; the contribution due to p can become significant for high degrees of multithreading. For the cache miss service time in a conventional processor substitute $p = 1$.

5 A Multithreaded Cache Model

We now derive the effective cache miss rate when p processor-resident threads share the cache. The cache model must account for the increase in cache miss rate as the number of processes grows. We assume a direct-mapped cache in our analyses. We will use the notation presented in Table 3, which is adopted from [28].

S	Direct-mapped cache size in terms of the number of cache sets (or rows)
B	Block size (same as network message length)
u	Working-set size (in blocks) of each process
τ	Size of the time interval used in measuring the working set
c	Collision rate used to compute intrinsic interference
v	Number of blocks a process leaves behind in the cache when it switches out, or the size of the <i>carry-over set</i>
m_{fixed}	Miss-rate component assumed fixed in our analysis

Table 3: Definition of terms used in the multithreaded cache model. In our analysis $\tau = 10,000$ and $c = 1.5$.

We start with a model for multiprogrammed caches. Such a model was derived by Thiebaut and Stone [29] for two processes, and a similar model for an arbitrary number of processes was derived by Agarwal, Horowitz, and Hennessy [28]. The model in [28] makes the following assumptions:

- Addresses have a uniform probability of mapping to any cache set.
- Processes obey the working-set model of program behavior [30], i.e., during any given time interval, a small set of blocks is in active use, and that the size of this set is u . Considerable empirical evidence exists in support of this model [31, 32, 33, 34]. For example, Belady and Kuehner [31], Kobayashi and MacDougall [32], and Thiebaut [33] have observed that the number of unique data blocks referenced by a program grows as some power function of time. If r denotes the time since the program started executing, the total number of

unique blocks can be represented as ar^b . Differentiating with respect to r , we obtain the rate at which new blocks are being referenced at time r as abr^{b-1} . Typical values reported for a range from about 1 to 10, and for b are about 0.5. When r becomes large (in the steady-state), it is clear that the rate of addition of new blocks is very small. Thus, the number of blocks in active use in the steady state (u) can be treated as a constant because not only is the rate of addition of new blocks very small, but some fraction of program blocks are also becoming inactive. In our address traces, we have found that an interval of $\tau = 10,000$ is sufficient to measure u , the steady-state working-set size. Nevertheless, to account for possible variations with time, our results cover a spectrum of values for u .

- The multiprogramming models in [28] and [29] further assumed that the context-switch interval (t) is greater than the time period τ used in measuring the working-set size. This assumption was required so that the process could fetch its entire working set into the cache during its interval on the processor, and greatly simplified the analysis.

We will not make the last assumption in our analysis, since, by its very nature, a multi-threaded processor switches contexts over very short intervals of time. In fact, our analysis assumes that switches are forced on every cache miss, which yields $t = 1/m$. Clearly, one miss can hardly replenish the entire working set of a process in the presence of several intervening processes. Therefore, only a portion of a process's working set can be retained in the cache in the steady state.

As an intuitive example, let us suppose context switches happen every cycle. Let the number of processes resident on the processor be large enough that a returning process i finds *none* of its blocks in the cache. As previously defined, the carry-over set of a process i is the set of blocks in its working set left in the cache when it switches out. On a cache miss, process i fetches exactly one block into the cache, yielding a steady-state carry-over set size of one, irrespective of its actual working-set size.

The rest of this section estimates the *multithreaded* cache miss rate. We distinguish this miss rate from the *multiprogrammed* miss rate: the former applies to very short context-switching intervals, while the latter assumes context-switching intervals large enough to allow a process to completely replenish its working set. We show in [21] that the two miss rates have a simple relationship: the multithreaded miss rate is greater than the multiprogrammed miss rate by a constant factor.

5.1 Review

Let us begin by reviewing relevant portions of the analytical cache model found in [28]. The *steady-state* cache miss rate is the sum of four components: nonstationary, intrinsic interference, multiprogramming, and coherence-related miss rates. The *nonstationary* component, denoted m_{ns} , is due to misses that bring blocks into the cache for the first time. The *intrinsic-interference* component, m_{intr} , results from the misses caused when the blocks associated with the working set of a given process interfere with each other in the cache. The *multiprogramming* component, $m_{cs}(p)$, arises from misses caused by blocks from different processes competing for cache residency. In a multiprocessor, *coherence-related invalidations* introduce an additional miss-rate component, m_{inv} . In this paper, we refer to the nonstationary and invalidation misses as the *fixed* miss-rate components and denote their sum as $m_{fixed} = m_{ns} + m_{inv}$. We derive the multithreaded miss rate by modifying the intrinsic and multiprogramming cache models.

Computation of the intrinsic-interference and multiprogramming miss-rate components requires the size of the carry-over set of a process. As previously defined, the size of the carry-over set of a process i is the number of blocks in the working set of process i left in the cache when it switches out. Let v denote the size of the carry-over set when the context-switch interval is large enough to allow complete replenishment of the carry-over set. With complete replenishment, v is independent of p . If blocks are mapped to cache sets uniformly with probability $1/S$, then v can be derived from u , the working-set size, as

$$v = S \left[1 - \left(1 - \frac{1}{S} \right)^u \right] \quad (6)$$

where the probability that a given block does not map into a given set is $(1 - 1/S)$, which when raised to the power u is the probability that no block from the process's working set maps into that set. One minus this quantity is the probability there is at least one block mapped to a cache set, which when multiplied by S yields the number of cache sets used. When $S \gg 1$, we can obtain the following simpler expression for v

$$v = S \left(1 - e^{-\frac{u}{S}} \right) \quad (7)$$

The situation is different in a multithreaded processor. As the context-switching interval decreases, so does the size of the carry-over set. Let $v'(p)$ be the *steady-state* carry-over set size with p processes sharing the cache for a small context-switch time. Multithreading the cache with a small interval can be modeled as an increase in both the intrinsic-interference and the multiprogramming components of the miss rate. A process effectively sees a smaller cache, which can be modeled by increasing the intrinsic-interference component to $m'_{intr}(p)$. Having to replenish part of the effective carry-over set each time the process is scheduled to run on the processor adds in the fine-grain context-switching component of misses denoted $m'_{cs}(p)$.

The analysis in the rest of this section focuses on models for the two cache miss-rate components, $m'_{cs}(p)$ and $m'_{intr}(p)$, when the context-switch interval becomes small.

5.2 Computing $m'_{cs}(p)$

We compute $m'_{cs}(p)$ by observing that the increase in the carry-over set size of a process is exactly one cache block when the process suffers a miss upon referencing a block displaced by an intervening process. Therefore, the steady-state value of the carry-over set size is reached when the $(p - 1)$ intervening processes displace exactly one cache block of process i from the cache for each block added by process i to its carry-over set.

We first require an estimate of the average number of distinct blocks referenced by a process in t cycles. Because a process does not fetch its entire working set during each context-switch interval, we have to model a steady-state situation, where the execution of some process j can affect another process i after several context-switch intervals of the process i . Accordingly, we estimate the average number of distinct blocks in the context-switch interval t from the average number of unique blocks in the larger time interval τ . Recalling that a process accesses u distinct blocks during time τ , we use tu/τ as an estimate of the average number of distinct references in context-switch time t . Another way of looking at this is to divide the time τ (over which the working set u was measured) into τ/t sub-intervals of length t , and to distribute the u unique references uniformly over these intervals.

Therefore, the number of unique references a process makes in t cycles is

$$tu/\tau$$

Similarly, when the $(p - 1)$ intervening processes run for t cycles each, we estimate the total number of unique blocks accessed by them as

$$(p - 1)t\frac{u}{\tau}$$

If the references of process i are randomly distributed throughout the cache, the probability that a referenced block is not already present in the cache is $(v - v'(p))/v$. The product of this probability and tu/τ , the number of unique blocks accessed by process i during interval t , yields the average number of previously displaced blocks fetched into the cache. Because we switch on every cache miss, this number is always less than or equal to one. Dividing by t , we derive $m'_{cs}(p)$ in terms of $v'(p)$ as

$$m'_{cs}(p) = \frac{v - v'(p)}{v} \frac{u}{\tau} \quad (8)$$

To obtain an expression for $v'(p)$, we will focus on the number of blocks displaced by the intervening processes. If the references of the intervening $p - 1$ processes are randomly distributed throughout the cache, then the probability that a given block of the intervening processes maps on top of a block of process i is $v'(p)/S$. The product of $v'(p)/S$ and the number of distinct blocks accessed by the $(p - 1)$ processes, $(p - 1)t\frac{u}{\tau}$, approximates the number of blocks of process i displaced from the cache. If we divide this number by t , we have an alternate expression for $m'_{cs}(p)$ given by

$$m'_{cs}(p) = \frac{v'(p)}{S} (p - 1) \frac{u}{\tau} \quad (9)$$

A more accurate – but complicated – expression for the number of displaced blocks is provided in [21] for large values of t . In practice, however, the complicated expression gives almost identical results to the simpler expression shown here. We now determine the steady-state carry-over set size $v'(p)$, and hence $m'_{cs}(p)$, from Equations 8 and 9 as

$$v'(p) = \frac{v}{1 + v\frac{(p-1)}{S}} \quad (10)$$

We make several useful observations from the above estimate of the carry-over set size of each process in a multithreaded processor. When the cache is very large ($S \gg u$), we find that $v'(p) \approx v$, and $v \approx u$, indicating that the cache can comfortably hold the entire working set of every process. When $S = v$, the effective size of the cached working set of each process is v/p . That is, each process gets only a fraction of its working set into the cache. Finally, when $S \ll v$, the size of the cached working set is limited to the cache size divided by the number of processes, or $S/(p - 1)$.

5.3 Computing $m'_{intr}(p)$

The interference component of misses in a multithreaded cache increases because each thread sees an effectively smaller cache. We compute this increase by modifying the cache model for intrinsic

interference in single-processor caches. The equation for the intrinsic miss-rate component in a direct-mapped cache derived in [28] is

$$\begin{aligned}
m_{intr} &= \frac{c}{\tau} \left[u - S \operatorname{bin} \left(u, \frac{1}{S}, d = 1 \right) \right] \\
&= \frac{c}{\tau} \left[u - u \left(1 - \frac{1}{S} \right)^{(u-1)} \right] \\
&\approx \frac{c}{\tau} \left(u - ue^{-\frac{u}{S}} \right)
\end{aligned} \tag{11}$$

where the term $\operatorname{bin}(u, \frac{1}{S}, d)$ is the binomial distribution,³ and represents the probability that d blocks from the working set of size u map into one of the S cache sets. If multiple blocks map to a cache set, they are called colliding blocks. For $d = 1$, the value of this distribution times S yields the number of blocks that do not collide with each other in the cache. By subtracting this number from u , we obtain the number that collide. The number of colliding blocks times the collision rate divided by the interval τ yields the miss rate. The collision rate, c , is a program dependent parameter that is independent of the cache size (for $S > u$). It indicates the average number of misses resulting from a colliding block.

In a multithreaded cache, the number of colliding blocks is different from the number computed in Equation 11. Because the size of the carry-over set of a process in a multithreaded cache is smaller than the size in a multiprogrammed cache ($v'(p) \leq v$), the number of colliding blocks in a multithreaded cache is greater. (Recall that the size of the carry-over set of a process is the average number of blocks of that process resident in the cache when the process switches out.) Given random placement of blocks in the cache, the number of noncolliding blocks in the multithreaded cache will decrease by the same fraction as the carry-over set, that is, by the fraction $v'(p)/v$. Thus, we can estimate the number of blocks that do not collide with other blocks of process i as

$$u \left(1 - \frac{1}{S} \right)^{u-1} \frac{v'(p)}{v} \approx ue^{-\frac{u}{S}} \frac{v'(p)}{v}$$

Substituting this value in Equation 11, the intrinsic interference component of the miss rate with p processes becomes

$$m'_{intr}(p) = \frac{c}{\tau} \left(u - ue^{-\frac{u}{S}} \frac{v'(p)}{v} \right) \tag{12}$$

5.4 Multithreaded Cache Model Summary and Simplifications

The net increase in the miss rate due to multithreading, $m'(p)$, is the sum of $m'_{cs}(p)$ and $m'_{intr}(p) - m_{intr}$, or

$$\begin{aligned}
m'(p) &= m'_{cs}(p) + m'_{intr}(p) - m_{intr} \\
&= v'(p) \frac{(p-1)u}{S} \frac{1}{\tau} + \frac{c}{\tau} ue^{-\frac{u}{S}} \left(1 - \frac{v'(p)}{v} \right)
\end{aligned} \tag{13}$$

³ $\operatorname{bin}(u, \frac{1}{S}, d) = \binom{u}{d} \left(\frac{1}{S} \right)^d \left(1 - \frac{1}{S} \right)^{u-d}$.

where v and $v'(p)$ are obtained from Equations 6 and 10 respectively. The overall cache miss rate can be represented as the sum of the fixed miss-rate component m_{fixed} , the single-process interference component (Equation 11), and the component due to multithreading (Equation 13), i.e.,

$$m(p) = m_{fixed} + m_{intr} + m'(p) \quad (14)$$

We will simplify the cache model to obtain the nature of the dependence of the miss rate on the number of processes. To exclude extraneous effects we will normalize the multithreading component to m_{intr} . Replacing $v'(p)$ and v by their respective formulae in terms of u , S , and p , and making the approximation $e^{-u/S} \approx (1 - u/S)$ when $u \ll S$, the ratio of the multithreading miss rate to the intrinsic miss rate simplifies to

$$\frac{m'(p)}{m_{intr}} \approx \frac{(p-1) \left(1 + \frac{1}{c}\right)}{1 + (p-1)\frac{u}{S}} \quad (15)$$

Furthermore, in the region where $(p-1)u \ll S$, we can further simplify the ratio to

$$\frac{m'(p)}{m_{intr}} \approx (p-1) \left(1 + \frac{1}{c}\right) \quad (16)$$

where m_{intr} can be expressed as

$$m_{intr} \approx \frac{c u^2}{\tau S} \quad (17)$$

Simulations suggest that the simple linear relationship between $m'(p)$ and p predicted by Equation 16 remains valid for up to about 10 processes.

6 Summary and Validation of the Model

The expressions derived for cache miss time in Equation 5 and for the cache miss rate in Equation 14 can be substituted into Equation 1 for the processor utilization. Table 4 summarizes these equations. Definitions of variables used in these equations are given in Tables 1, 2 and 3.

We conducted several experiments to verify that our approximations were indeed valid. The cache model is validated in this section and the network model is validated in [27]. We present additional evidence of the accuracy of the overall multithreaded processor model by comparing it to published simulation results in Table 6 in Section 8.

Cache model validations compare predicted miss rates with the miss rates obtained through trace-driven simulations of caches [35] using multithreaded address traces. In the absence of traces obtained from real multithreaded processors, we synthesize multithreaded traces in two ways. First, references from processor i in a multiprocessor trace are assigned to a context i on a multithreaded processor, creating a situation where processes that ran concurrently on separate processors execute as multiple contexts on one processor instead. Second, we extract a trace of one processor from the multiprocessor trace and replicate it multiple times to simulate the effect of an interleaved multithreaded-processor trace. Let us refer to the first set of traces as *multithreaded traces*, and to the second set of traces as *replicated traces*. In both methods, each single-thread trace is roughly 300,000 references long, and the multithreaded or replicated traces are longer by a factor p , where p is the number of hardware contexts. A random, distinct process

<p>The multithreaded processor utilization:</p> $U(p) = \begin{cases} \frac{p}{1+T(p)m(p)} & \text{for } p < \frac{1+T(p)m(p)}{1+Cm(p)} \\ \frac{1}{1+Cm(p)} & \text{for } p \geq \frac{1+T(p)m(p)}{1+Cm(p)} \end{cases}$ <p>The network model:</p> $T(p) = \frac{T_0}{2} + \frac{Bpk}{6} - \frac{1}{2m(p)} + \frac{1}{2} \sqrt{\left(T_0 - \frac{Bpk}{3} + \frac{1}{m(p)}\right)^2 + 8pB^2n\frac{k}{3} \left(1 - \frac{3}{k}\right)}$ $T_0 = 2n\frac{k}{3} + M + B - 1$ <p>The cache model:</p> $m(p) = m_{fixed} + m_{intr} + m_{intr} \frac{(p-1)\left(1+\frac{1}{c}\right)}{1+(p-1)\frac{u}{S}}$ $m_{intr} = \frac{c}{\tau} \frac{u^2}{S}$
--

Table 4: Summary of the model.

identifier (PID) is assigned to each thread in the interleaved trace. The PID of each thread is hashed into the address, and the cache is indexed with the resulting hashed value to avoid systematic collisions with the same address of the other processes. The PID hashing scheme has also been suggested as a way of improving the performance of virtual-address caches [34].

While the multithreaded traces are more realistic, the replicated traces are easier to use in validation experiments because they can be created with unlimited, varying numbers of threads that have the same statistical properties. The maximum number of processors in the multiprocessor trace limits the number of contexts we can simulate with a multithreaded trace (although it is possible to synthesize a replicated multithreaded trace). Because multiprocessor traces are hard to obtain, replicated traces can use single-processor traces. Furthermore, many of our traced programs follow the Single-Program-Multiple-Data Model, where the individual processors run the same code but operate on different data sets. With this programming model, the memory reference behavior within individual processor traces is expected to be uniform across all processors, and a replicated trace with randomized process identifiers is a fair representation of this.

The cache model validations use the following traces:

LocusRoute: An eight-processor, physical-address trace of a parallel global router for VLSI standard cells. The trace was obtained using the VAX trap bit with round-robin scheduling of processes.

ParaOPS5: A four-processor, physical-address trace of a parallel implementation of the OPS5 rule-based language.

PTHOR: A four-processor, physical-address trace of a parallel logic simulation run.

PTHOR-V: A four-processor, virtual-address trace of a logic simulation run. ParaOPS5 and PTHOR were traced using a microcode-based multiprocessor tracing scheme called ATUM [36] on a VAX 8350.

SIMPLE: A 64-processor, virtual-address trace of a program modeling hydrodynamic and thermal behavior of fluids in two dimensions. The trace was created by a post-mortem

scheduling scheme at IBM.

IVEX: A single-processor, virtual-address trace of a DEC program, Interconnect Verify, checking net lists in a VLSI chip (under VMS).

PASC: A single-processor, virtual-address trace of a PASCAL compile of a microcode parser program. IVEX and PASC were obtained using single-processor ATUM on a VAX 8350.

Replicated traces with varying numbers of contexts were synthesized using single-process traces from each of the above traces. Figure 2(a) compares model predictions and simulation results from the resulting interleaved traces, and Figure 2(b) shows the aggregate miss rates. We also carried out experiments with multithreaded traces of SIMPLE, LocusRoute, and ParaOPS5. The multithreaded miss rates of these traces were largely indistinguishable from those of the corresponding replicated traces. Because of limitations in the number of processors in the multithreaded traces, ParaOPS5 and LocusRoute were simulated with a maximum of four and eight contexts respectively. We did not use PTHOR for the multithreaded trace experiments, because several processes constituted each processor trace, and we did not want to wrestle with a multithreaded multiprogrammed cache model.

The curves in Figure 2(a) show the increase in miss rate due to multithreading for caches with $S = 16K$ and $B = 4$.⁴ Each point in Figure 2(a) represents the average miss-rate increase over ten simulation runs. Multiple simulation trials were conducted to reduce the effect of statistical variations in the hashing of addresses from various processes into cache sets. The predictions for the individual traces as well as the average over all traces were quite good. PASC, LocusRoute, and SIMPLE display smaller miss-rate increases with the number of processes because their working-set sizes are smaller in comparison with those of IVEX, ParaOPS5, and PTHOR. Part of the reason IVEX, ParaOPS5, and PTHOR traces have a higher working set size is that, unlike the other traces, they include operating system references captured by the ATUM tracing scheme.

We found that the approximate miss-rate model using Equation 16 is valid up to only about 10 processes, which is expected because the approximation is valid only when $(p - 1)u/S \ll 1$. The more accurate Equation 15 is virtually indistinguishable from the most accurate model of Equation 13 (shown in the Figure).

7 Performance Implications of Multithreading

Multithreading impacts performance in several ways. Typically, as the degree of multithreading increases, so does the network latency and the cache miss rate. A higher degree of multithreading allows some fraction of the network delay to be overlapped with computation, often increasing the processor utilization. However, network-bandwidth limitations and the context-switching overhead may limit the achievable utilization. Let us first make some general observations from the model.

⁴For the purpose of validation, the correction to the model for nonstationarity in the program [21] is included in the graphs. Because of systematic collisions in smaller caches, the SIMPLE graphs use a cache with $S = 64K$.