LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-501

PERFORMANCE TRADEOFFS IN MULTITHREADED PROCESSORS

Anant Agarwal

April 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

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. Large caches yield close to full processor utilization with as few as two to four contexts, while small caches may require up to four times as many 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 of 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].

MIT VLSI Memo, 1989 No. 89-566; revised 1990.
Submitted to IEEE Transactions on Parallel and
Distributed Systems, 1991.

There are limits to the improvements in processor utilization achievable by multithreading a processor. Most important, multithreading requires applications to display sufficient parallelism so that multiple threads can be assigned to each processor.[1] 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. When the network is saturated, increased network latencies will decrease the benefits of multithreading. 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.

We derive a simple expression for processor utilization that takes into account the effects of network contention, cache interference, and context switching. As an indication of the results we obtained, for the set of default parameters given in Table 5, we found that when the number of processor-resident threads is increased from one to two the processor utilization increases from 0.4 to 0.7. However, due to increased cache interference and context-switching overhead, processor utilization only increased from 0.7 to 0.8 when the number of processor-resident threads went from two to three. We also show that high processor efficiencies can be achieved in coarsely 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, register file management, impact on the clock cycle of register file size, and context-switch decision making. 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 associated with the Alewife multiprocessor project (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. Furthermore, driving the model with parameters measured from real parallel applications does lend credibility to our results.

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

[1]This paper uses the terms process, thread, task, and context equivalently.

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 [13], and remote memory transactions are satisfied over a direct mesh interconnection 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 listed here: (1) the processor must have a low context-switching overhead, and (2) the processor must achieve high single-thread performance. As shown in Section 7, the 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 efficiently run sections of applications with low parallelism.

Unfortunately, the above goals are hard to achieve simultaneously. Obtaining very small overheads (say less than three cycles) without compromising single-thread performance can significantly complicate the processor. Multiple instructions from the same process must be allowed in consecutive stages of the processor pipeline to achieve high single-thread performance, which not only increases the amount of state that must be saved and restored on a process switch, but also increases the difficulty of cleanly halting the pipeline. For example, if a remote memory operation occurs, successive instructions in the pipeline must be stalled (however, special support in the form of register full/empty bits [2] can reduce the need to stall 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. However, the effect of pipeline flushing is harder to minimize unless a mechanism to speedily save and restore the entire pipeline state (e.g., bypass registers, decoded instructions, fetched register contents) is implemented. Additional functionality to mitigate these problems, such as special frames for pipeline state, might adversely impact the processor cycle time.

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 [14], 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 four 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, and the second frame is used 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 [15] modified to support coarse multithreading is called SPARCLE. The register set is divided into several frames that are conventionally used as register windows [16, 17] 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. Several studies have shown that single process frames, combined with register allocation methods, can achieve comparable performance to register windows (e.g., see [18, 19]). Our hardware modifications will 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 [13]. Adopting an integrated-systems approach, the hardware implements a small amount of coherence directory state to handle the common case of limited sharing of data blocks. The controller interrupts the processor for software extension of the directory into local memory when a directory overflow occurs. The controller also relegates the responsibility of handling exceptional situations, such as special message arrival or network buffer overflow, to the processor, which significantly simplifies the controller design. Rapid context switching makes the use of coherence protocols that guarantee sequential consistency feasible because the processor can switch to a different thread while the acknowledgments to outstanding memory transactions are awaited. Rapid trap handling also allows efficient handling of Futures [20] and full/empty bit [2] traps.

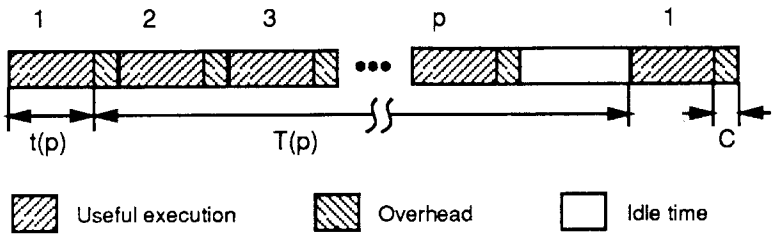

Figure 1: Hiding network latency by multithreading the processor.

3 A Multithreaded Processor Performance Model

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), 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 also 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 the 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 $pt(p)$. Otherwise, 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 becomes,[2]

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

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 in Table 1 the terms used thus far, 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 processor utilization equation.

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 invalidations. The multithreaded cache model computes the cache miss rate as a function of the degree of processor multithreading p. 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

[2] If 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

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

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

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

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, and 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 Appendix A.1 to account for sharing effects by reducing the size of the individual process working sets by the shared fraction.

- Our results do not include the effect that the non-stationary behavior of the process has on the cache. A process suffers non-stationary 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 this effect is not accounted for, the analysis will incorrectly associate some cache misses with multithreading rather than with the non-stationary behavior of the program. Even though we have found that this effect is not significant for most applications, Appendix A.2 modifies the model to account for this effect. It is interesting to note that applications that suffer from a high non-stationary miss rate component (represented as m_{ns} in our model) will have a relatively low multithreading related miss rate component, and errors in estimating the latter component will not significantly impact the results.

- Our processor utilization model assumes a fixed time interval $t(p)$ between context switches and a fixed cache miss service time $T(p)$. We could also 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 $T(p)$. Due to the loop nature of many programs, exponential time between misses is hard to justify, and with lightly loaded networks the network response times will be fairly uniform. Nevertheless, the processor utilization can be derived from a simple M/M/1//M queueing model [21] for a finite population of size p, a single queueing server (the processor) with service times that are exponentially distributed, and 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 the queueing server (the processor) is idle, or

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

  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 effect of the added p term in the denominator is to generally lower 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 [22] 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, and messages are routed completely in one dimension before the next. If an indirect network [23] is used instead, the model described in Appendix D could be applied. 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 and less scalable. Our network analysis assumes:

- Infinite buffering at the switch nodes. Simulation experiments [24] have shown that as few as four packet buffers at each switch node can approach infinite buffer performance.
- Uniform traffic rates from all the nodes.
- Uniformly distributed and independent message destinations.

The network parameters are summarized in Table 2. The message size is B times the network channel width, the network dimension is n, the radix is k, and the memory access time is M cycles. 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 a high-radix, buffered, k-ary n-cube direct network is derived and validated in [25]. Appendix C summarizes the derivation and validation of this model. From Appendix C, the average cache miss service time T is given by,

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

where ρ is the network channel utilization, B is the message size, and k_d is the average distance traveled by a message in a given 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. 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 queueing delay.

Assuming separate channels for both directions, with random message entry and exit points in each dimension, the average distance in a dimension can be computed as

$$k_d = \frac{k}{3}$$

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 $2n$ associated channels (separate channels for each direction exist), the channel capacity consumed, or the channel utilization, is given by,

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

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

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

The above expression implies that the network latency 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 processes share the cache. The cache model must account for the increase in cache miss rate as the number of processes grows. The notation presented in Table 3 is adopted from [26].

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 *carry-over set*
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 [27] for two processes, and a similar model for an arbitrary number of processes was derived by Agarwal, Horowitz, and Hennessy [26]. The model in [26] makes the following assumptions:

- The mapping of addresses to cache sets is random.
- Processes obey the working-set model of program behavior [28], 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 (e.g., [29, 30, 31]). For example, Kobayashi and MacDougall [29] and Thiebaut [30] 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 the steady-state working-set size u. Nevertheless, to account for possible variations with time, our results cover a spectrum of values for u.

- The multiprogramming models in [26] and [27] 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 multithreaded 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 fraction 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 processors 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 for very short context-switching intervals. For comparison, Appendix B presents a simplified *multiprogrammed* cache miss rate model for direct-mapped caches with context-switching intervals large enough to allow a process to completely replenish its working set. Interestingly, we discover that the two miss rates have a simple relationship – the multithreaded miss rate is greater than the multiprogrammed miss rate by a constant factor. Next we review relevant portions of the analytical cache model found in [26].

5.1 Review

The *steady-state* cache miss rate is the sum of four components: non-stationary, intrinsic interference, multiprogramming, and coherence-related miss rates. The *non-stationary* 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 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 non-stationary and multiprocessor 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.

The equation for the intrinsic miss rate component in a direct-mapped cache derived in [26] is:

$$\begin{aligned} m_{intr} &= \frac{c}{\tau}\left[u - S\, bin\left(u, \frac{1}{S}, d = 1\right)\right] \\ &= \frac{c}{\tau} u\left[1 - \left(1 - \frac{1}{S}\right)^u\right] \end{aligned} \tag{6}$$

where c, the collision rate, is independent of cache size (for cache sizes greater than the working-set size) and is treated as a constant in our analysis. The term $bin(u, \frac{1}{S}, d)$ 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. For $d = 1$, u minus S times the value of this distribution yields the number of blocks of the process that collide with each other in the cache. If multiple blocks map to a cache set, they are called colliding blocks. The number of colliding blocks times the collision rate divided by the interval τ yields the miss rate.

Computation of the multithreaded miss rate requires the size of the carry-over set of a process, denoted as v. As previously defined, the size of the carry-over set of a process i is the number of of blocks in the working set of process i left in the cache when it switches out. If the context-switch interval is large enough to allow complete replenishment of the carry-over set, v is independent of p. If blocks are randomly mapped to cache sets, then the carry-over set can be derived from the working set size u as

$$v = S\left[1 - \left(1 - \frac{1}{S}\right)^u\right] \tag{7}$$

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. The multiprogramming miss rate component $m_{cs}(p)$ is derived by counting the difference between v and the number of blocks of the process remaining in the cache when the process returns after $p - 1$ intervening processes. See Appendix B for the complete multiprogramming model.

5.2 Miss Rate with Small Context-Switch Intervals

The analysis in this section focuses on modified models for two cache miss rate components, $m_{cs}(p)$ and $m_{intr}(p)$, when the context switch interval becomes small. Let $m'(p)$ denote the net increase in the miss rate due to multithreading. As the context-switching interval decreases, the size of the carry-over set also reduces. 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 increases the intrinsic interference component of misses. The process effectively

[3] $bin(u, \frac{1}{S}, d) = \binom{u}{d}\left(\frac{1}{S}\right)^d\left(1 - \frac{1}{S}\right)^{u-d}$. Also, $u - d \approx u$ for typical values of u and d.

sees a smaller cache and hence its intrinsic interference component increases 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 $m'_{cs}(p)$.

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

We will first derive the component $m'_{cs}(p)$. Let $v'_j(p)$ be the carry-over set size before steady state is reached for the j^{th} occurrence of process i on the processor, and let t be the context-switch interval. We first compute $v'(p)$ and then obtain $m'_{cs}(p)$.

We first require an estimate of the average number of distinct blocks referenced by a process during its context-switch interval of t cycles in order to determine the steady-state interference between threads. 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 interval 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. Then, a process has tu/τ unique references within each sub-interval. Similarly, the total number of unique blocks accessed by the intervening processes can be estimated as

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

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 $v'_j(p)/S$. From this we can approximate the number of blocks of process i displaced from the cache as

$$\frac{v'_j(p)}{S}(p-1)t\frac{u}{\tau}$$

The above expression assumes that the probability of purging a block of process i stays constant throughout the intervening $(p-1)$ processes. An accurate expression for the purged blocks is $v'_j(p)[1-(1-1/S)^{(p-1)tu/\tau}]$ if t is large enough that a sizable fraction of the carry-over set is purged. In practice, however, the simpler expression gives almost identical results to the more complicated expression.

Similarly, when the process i is resumed, it replenishes some fraction of its set of blocks in the cache. If it runs for t cycles, accessing tu/τ unique blocks during this time, the number of previously-displaced blocks fetched into the cache is

$$\frac{v-v'_j(p)}{v}t\frac{u}{\tau}$$

where $(v-v'_j(p))/v$ is the probability that a referenced block is not already present in the cache. As before, a more accurate expression for the number of new blocks can be derived as $(v-v'_j(p))[1-(1-1/v)^{tu/\tau}]$ when t is large.

In the steady state, the number of purged blocks must equal the number of replenished

blocks, and $v'_j(p) = v'(p)$. Thus,

$$\frac{v'_j(p)}{S}(p-1)t\frac{u}{\tau} = \frac{v - v'_j(p)}{v}t\frac{u}{\tau}$$

Simplifying, the steady-state carry-over set size with a multithreading degree of p is,

$$v'(p) = \frac{v}{1 + v\frac{(p-1)}{S}} \tag{8}$$

The above equation indicates that for context-switching intervals that are small compared to τ, the carry-over set size is independent of t. For large t, when the probability of purging or replenishing a block in the carry-over set is no longer constant over the time interval between occurrences of a process, the carry-over set-size does indeed depend on t, and the more complex expressions shown earlier, or their higher order approximations, must be used.

We can make several useful observations from the above estimate of the carry-over set size $v'(p)$ of each process in a multithreaded processor. When the cache is very large ($S >> u$), 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 cached working set of each process is v/p. That is, each process gets only a fraction of its working set in the cache, and the size of the cached working set is inversely proportional to the number of processes. Finally, when $S << v$, the cached working set is limited to the cache size divided by the number of processes, or $S/(p-1)$.

The corresponding miss rate due to context switching ($m'_{cs}(p)$) is the rate at which blocks in the carry-over set of a process are purged by intervening processes.

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

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

Now, let us compute the increase in the intrinsic interference component. The number of colliding blocks is different from the number computed in Equation 6. The new number is simply the difference between the working-set size u and the number of blocks in the carry-over set $v'(p)$ that do not collide with other blocks of the process i. Given random placement of blocks in the cache, the number of non-colliding blocks in the multithreaded cache will decrease by the same fraction as the carry-over set, that is, by the fraction $v'(p)/v$. We can thus estimate the number of blocks that do not collide with other blocks of process i as

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

Substituting this value in Equation 6, the intrinsic interference component of the miss rate with p processes becomes,

$$m'_{intr}(p) = \frac{c}{\tau}\left[u - u\left(1 - \frac{1}{S}\right)^u \frac{v'(p)}{v}\right] \tag{10}$$

We now derive the increase in the intrinsic interference component due to multithreading as,

$$m'_{intr}(p) - m_{intr} = \frac{c}{\tau} u \left(1 - \frac{1}{S}\right)^u \left(1 - \frac{v'(p)}{v}\right) \tag{11}$$

Intuitively, in the above equation, $u(1 - 1/S)^u$ is the number of blocks of a processes that do not collide in a single-process cache, $(1 - v'(p)/v)$ is the fraction of these blocks that become involved in collisions in a multithreaded cache, and c/τ is the average number of times a colliding cache block causes cache misses.

Thus, 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,

$$\begin{aligned} m'(p) &= m'_{cs}(p) + m'_{intr}(p) - m_{intr} \\ &= v'(p)\frac{(p-1)}{S}\frac{u}{\tau} + \frac{c}{\tau} u \left(1 - \frac{1}{S}\right)^u \left(1 - \frac{v'(p)}{v}\right) \end{aligned} \tag{12}$$

where v and $v'(p)$ are obtained from Equations 7 and 8 respectively. The overall cache miss rate can be represented as the sum of the fixed miss rate component m_{fixed}, single process interference component (Equation 6), and the component due to multithreading (Equation 12).

$$m(p) = m_{fixed} + m_{intr} + m'(p) \tag{13}$$

5.3 Multithreaded Cache Model Summary and Simplifications

The cache model can be simplified to obtain the nature of the dependence of the miss rate on the number of processes. We will first obtain a simplified expression for the ratio of $m'(p)$ and m_{intr}. Replacing $v'(p)$ and v by their respective formulae in terms of u, S, and p, and making the approximation

$$\left(1 - \frac{1}{S}\right)^u \approx \left(1 - \frac{u}{S}\right)$$

when $u << S$, the ratio of the multithreading induced miss rate to the intrinsic miss rate simplifies to:

$$\frac{m'(p)}{m_{intr}} \approx \frac{(p-1)\left(1 + \frac{1}{c}\right)}{1 + (p-1)\frac{u}{S}} \tag{14}$$

This yields for the overall miss rate,

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

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

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

The overall miss rate is then,

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

where, as before,

$$m_{intr} = \frac{c}{\tau}\left[u - u\left(1-\frac{1}{S}\right)^u\right]$$

or simplified for small $u << S$,

$$m_{intr} \approx \frac{c}{\tau}\frac{u^2}{S} \tag{17}$$

We have verified through simulations that the simple linear relationship between the multithreading miss rate and the number of processes on the processor predicted by Equation 16 is remarkably accurate for up to about 10 processes for the parameters shown in Table 5.

6 Summary and Validation of the Model

The expressions derived for network latency in Equation 5 and for the cache miss rate in Equation 13 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.

The multithreaded processor utilization:

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

The network model:

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

$$T_0 = 2n\frac{k}{3} + M + B - 1$$

The cache model:

$$m(p) = m_{fixed} + m_{intr} + m_{intr}\frac{(p-1)\left(1+\frac{1}{c}\right)}{1+(p-1)\frac{u}{S}}$$

$$m_{intr} = \frac{c}{\tau}\frac{u^2}{S}$$

The variables for the processor utilization, network, and cache models are defined in Tables 1, 2 and 3 respectively.

Table 4: Summary of the model.

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 Appendix C. 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 [32] of caches 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 on 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 identifier is assigned to each thread in the interleaved trace. The process identifier 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 [31].

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 a real multithreaded trace.

The cache model validations use the following traces:

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

ParaOPS5: A parallel implementation of the OPS5 rule-based language.

PTHOR: A physical address trace of a parallel logic simulation run.

PTHOR-V: A virtual address trace of the logic simulation run. ParaOPS5 and PTHOR were traced using a microcode-based multiprocessor tracing scheme called ATUM [33] on a four-processor VAX 8350.

SIMPLE: A program modeling hydrodynamic and thermal behavior of fluids in two dimensions running on 64 processors traced by a post-mortem scheduling scheme at IBM.

IVEX: A DEC program, Interconnect Verify, checking net lists in a VLSI chip (under VMS).

PASC: PASCAL compile of a microcode parser program. IVEX and PASC are single-processor traces 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 ParaOPS. 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, ParaOPS and LocusRoute were simulated with a maximum of four and

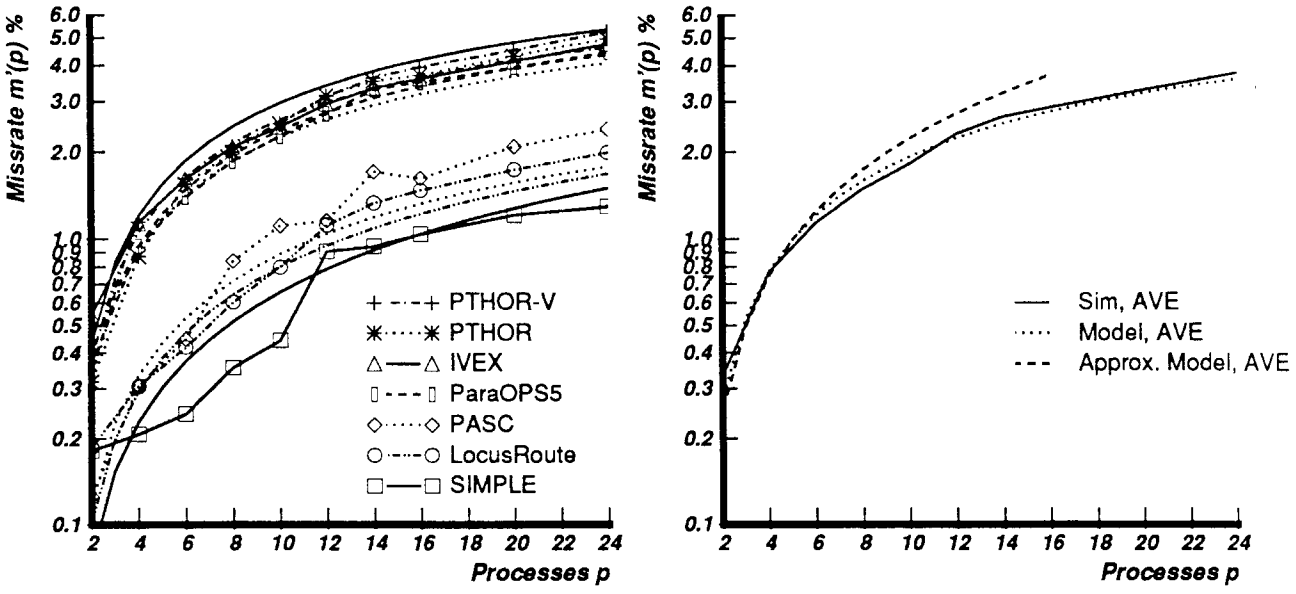

Figure 2: Increase in the miss rate of a multithreaded cache. (a) Individual trace results. Curves with symbols correspond to simulations and those without symbols correspond to the model. (b) Averages over all traces. The curve for the approximate model is plotted in the range $(p-1)u << S$.

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. The miss rate increase with number of processors is small in PASC, LocusRoute, and SIMPLE because of their smaller working set sizes compared to IVEX, ParaOPS5, PTHOR, and PTHOR-V. Part of the reason IVEX, ParaOPS5, and the 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 15 is valid up to only about 10 processes, which is expected because the approximation is invalid when $(p-1)u/S$ is no longer negligible compared to 1. The more accurate Equation 14 is virtually indistinguishable from the most accurate model of Equation 13 (shown in the Figure).

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

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.

7.1 General Observations

Multithreading imposes higher bandwidth requirements on the network. As the network approaches capacity, the increase in network latency is proportional to the number of processes. When the available network bandwidth is saturated, adding more processes will not improve processor utilization. For example, let us calculate the limits on processor utilization for the default set of system parameters, given that the cache miss rate is fixed at 4%. If the available network bandwidth corresponds to a channel utilization of $\rho = 0.7$, the maximum number of processes the network can sustain at a node is 2 (using $\rho = pBk_d/(t+T)$, from Equation 4). The corresponding network-bandwidth-limited utilization is 0.64 (from $U = p/(1+mT)$).

A network with a higher latency for the same bandwidth would require a higher degree of multithreading to achieve the same processor utilization. For example, suppose each network switch had a transit time of s cycles (we used $s = 1$ thus far). The greater switch transit time increases the fixed network delay (from Table 4) to $T_0 = 2snk/3 + M + B - 1$. A similar effect is achieved by increasing the memory latency M. If $s = 4$, the maximum number of usable processes is 5, and the corresponding network-bandwidth-limited utilization is 0.64, as before.

A higher processor utilization can be achieved by increasing the available network bandwidth, say by using a faster network clock or making network channels wider.

From the simple model, the processor utilization does not change due to multithreading if the intrinsic interference component of the cache miss rate is high, which can happen if the cache is small compared to the working-set size of a process. This is easily seen from the processor utilization Equation 1 with $m(p)$ substituted from Equation 16. A high m_{intr} component implies that the processor utilization is roughly,

$$U(p) \approx \frac{p}{T(p)m_{intr}(p-1)\left(1+\frac{1}{c}\right)} \quad \text{for } p > 1$$

However, if the ratio of processes to cache size, $(p-1)/S$, is held constant, the utilization increases linearly with the number of processes, despite a high interference component. That is, to support more processes with the same high interference miss rate, the cache should be made proportionally larger, as can be seen by substituting $m_{intr} = cu^2/S\tau$ in the above equation,

$$U(p) \approx \frac{p}{T(p)\frac{cu^2}{S\tau}(p-1)\left(1+\frac{1}{c}\right)} \quad \text{for } p > 1$$

On the other hand, if the interference miss rate component is small compared to the fixed miss rate component, $m_{fixed} = m_{ns} + m_{inv}$, utilization will increase linearly with the number of

Parameter	Value
Context-switching overhead C	4 cycles
Memory access time M	10 cycles
Network dimension n	3
Network radix k	20
Fixed miss rate m_{fixed}	2%
Average message size B	4
Cache block size	16 bytes
Process working-set size u	250 blocks
Cache size	64K bytes

Table 5: Default parameter values used in the analyses.

processes given the same cache size. The utilization equation becomes

$$U(p) \approx \frac{p}{1 + T(p)m_{fixed}} \text{ for } p > 1$$

Our experience with multiprocessor simulations with parallel applications has been that the fixed miss rate component is generally large relative to the intrinsic interference component. Both higher multiprocessor cache invalidation rates and smaller working sets of fine-grain processes contribute to this observation, and future multiprocessors are increasingly likely to operate in this range.

For large caches, the cache model supports the intuition that the interference component of the miss rate is proportional to the number of processes and inversely proportional to the cache size.

The intrinsic and multithreading interference components of the miss rate are related to the square of the working-set sizes of the processes (see Equations 17 and 15). Working-set size changes will therefore dramatically impact cache performance. Hence effort put into compacting data words efficiently into cache blocks to increase the fraction of used words in cache lines will be effort well spent. Process sharing of blocks on a processor must be encouraged for the same reason. Reduction in the private working sets of the processes yields proportionally greater benefits in cache miss rate and processor utilization due to the square law dependence on the working set size.

As investigated in Appendix A.1, threads can benefit substantially by sharing data in the cache. Sharing helps in two ways: the working-set sizes that contribute to multithreading interference misses are effectively reduced, and the cost of fetching in shared blocks are amortized over all the threads.

Context-switching overhead becomes important only when the number of processes is large enough to completely overlap the network latency with processor execution. The reason is that for a smaller number of processes, the processor cycles wasted in context switching would otherwise have been spent in waiting on the network. The impact of the context-switching overhead is more significant when it approaches the size of the context-switching interval.

We now use the accurate model to estimate the effects of varying the parameters on processor utilization using the default parameters given in Table 5. The miss rate, working-set size, and message size defaults are typical of the applications we have measured [34]. Figure 3 shows the degree to which each component impacts overall processor utilization. The ideal curve shows

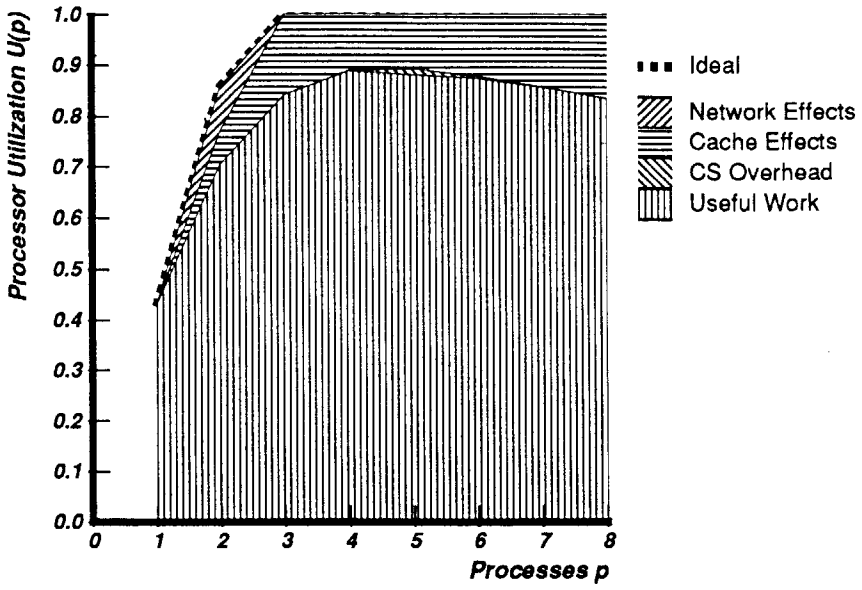

Figure 3: Relative sizes of the various components that affect processor utilization.

the increase in processor utilization when both the cache miss rate and network contention correspond to that of a single process, and do not increase with the degree of multithreading p. The curve denoting network effects includes the increase in network contention due to increasing p. The cache effects curve include the impact of both an increased cache miss rate and the corresponding increased network delays. It is easy to see that as the degree of multithreading is increased, the processor utilization can actually drop due to higher cache miss rates and network contention.

7.2 Effect of Network Contention

Let us first investigate the impact of the increased network contention on processor utilization keeping the other factors such as cache miss rate at a constant value and the context-switching overhead at zero. Figure 4 shows the effect of contention for several values of the network radix k in a three dimensional network ($n = 3$).

For small latencies, as in the case for a small multiprocessor system, one to three processes are sufficient to yield close to perfect utilization when the cache effects and overhead are ignored. As network latencies become larger in a large-scale machine, adding more processes is always beneficial, although the marginal benefit of each added process becomes small after a few processes due to the increased network contention.

7.3 Effect of Context-Switching Overhead

Figure 5 shows the effect of context-switching overhead for the default set of parameters. Context-switching overhead gives rise to a limiting utilization of $1/(1 + m(p)C)$. Once utilization reaches its maximum value for $p = (1 + m(p)T(p))/(1 + m(p)C)$, increasing it further decreases the utilization because the miss rate becomes worse. We see this effect for overheads 8, 16, and 32, in Figure 5. For smaller overheads, the network bandwidth is the constraining

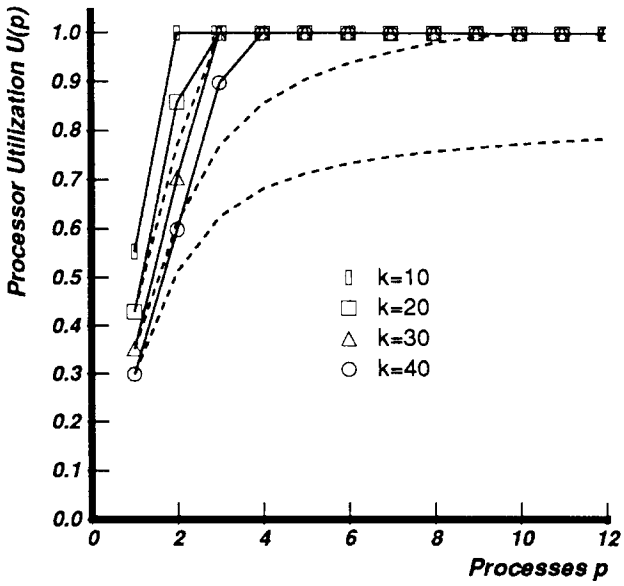

Figure 4: Impact of increased network contention on processor utilization. The solid curves represent the ideal utilization without the extra network contention and the dashed curves correspond to the lowered utilization due to network contention. For $k = 10$, the dashed curve is indistinguishable from the solid curve.

factor.

The effect of overhead will have a significant impact on processor design. As discussed in Section 2, obtaining very low-overhead context switches while retaining high single thread performance is not easy. If slightly higher overheads can be tolerated without compromising processor utilization, the task of the processor designer will be substantially simpler.

Higher overheads can be tolerated if the frequency of context switching is low. In a cache-based system, this frequency is related to the cache miss rate. Observe that with an eight cycle overhead, three processes can still yield up to 80% processor utilization. Put another way, because the product of the miss rate and the overhead determine utilization (see Equation 1), a relatively high context-switching overhead can be tolerated if the miss rate is correspondingly low.

7.4 Cache Effects

Figure 6 shows the effect of the fixed cache miss rate for the default parameter set. As few as two processes closely approach being able to fully mask network latency for fixed miss rates of up to 1%. The higher fixed miss rate of 2% requires four processes to obtain comparable utilization, although the utilization bears an almost linear relationship to the number of processes because of the low interference component. Recall that the multithreading interference overhead is determined by the product of the intrinsic interference component and the number of processes.

Figure 7 clearly shows the dramatic impact of working-set size on processor utilization. When the working-set size is large, processor utilization can decrease with the degree of multithreading. The reason is that the larger working set suffers a higher interference component, which causes

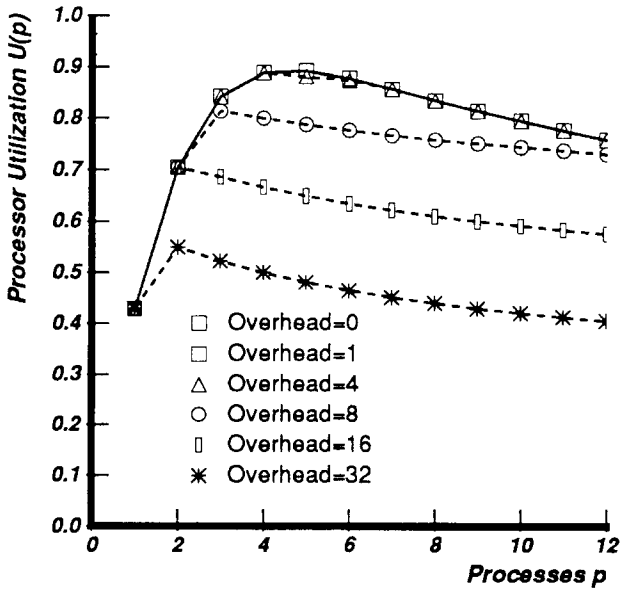

Figure 5: Impact of context-switching overhead on processor utilization. Overhead is specified in processor cycles.

the overall miss rate to increase in proportion to the number of processes.

Figure 8 assesses the impact of cache size keeping m_{fixed} the same for all caches. Cache size effects are small for caches greater than 64K bytes ($S = 4096$) because large caches can comfortably sustain the working sets of multiple processes (see discussion following Equation 8). In large caches the interference component is much smaller than the non-stationary component, making multithreading always useful. Smaller caches suffer more interference, and the relative benefit of increasing the number of processes is smaller. For example, two processes achieve 80% processor utilization in a 256K byte cache ($S = 16,384$), while a comparable utilization requires three processes in a 64K byte cache ($S = 4096$). The marginal benefit of adding more processes keeps decreasing as caches become smaller due to increased network contention.

7.5 Effect of Network and Memory Latency

How large a network and memory latency can we mask using multithreading? To answer this question, let us suppose we have 4 processes available to run on the processor. Then as the flat region of the corresponding curve in Figure 9 depicts, multithreading can mask network latencies in machines with network radices up to 20 yielding processor utilization of close to 90%. For this machine, if the number of available processes were to drop to 2, then the utilization would drop to 70%.

8 Related Work

Halstead advocates the use of multithreaded processors as the processing nodes in multiprocessors and presentes some analyses to estimate the benefits of multithreading [35], but without accounting for several important factors such as network contention and context-switching over-

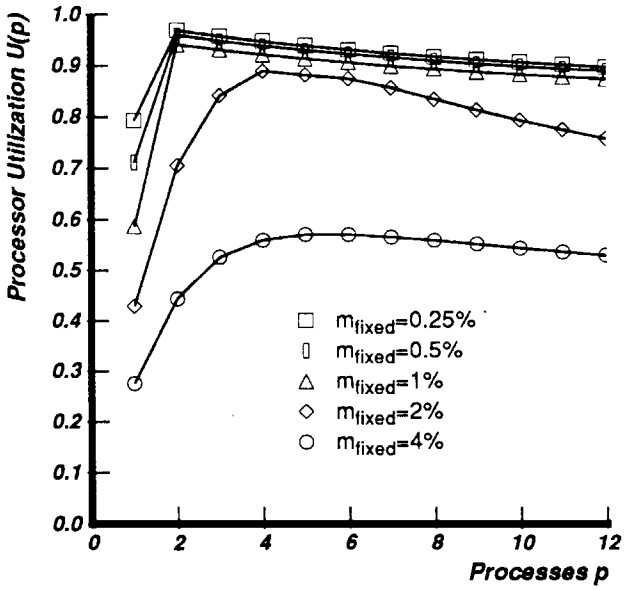

Figure 6: Impact of higher cache miss rates on the processor utilization. Varying the fixed cache miss rate component.

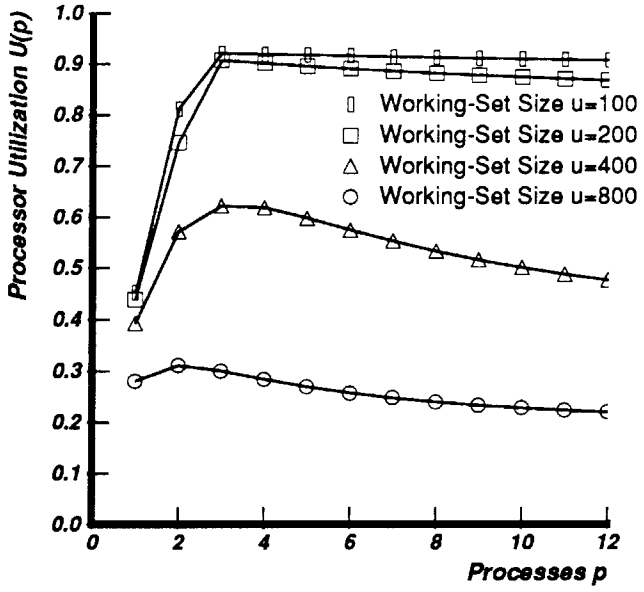

Figure 7: Impact of increased cache interference on processor utilization. Varying the working-set size.

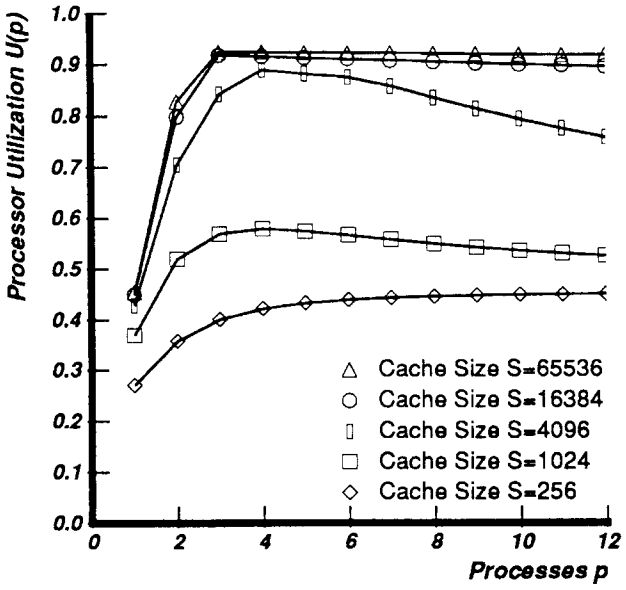

Figure 8: Impact of increased cache miss rates on processor utilization. Varying cache size.

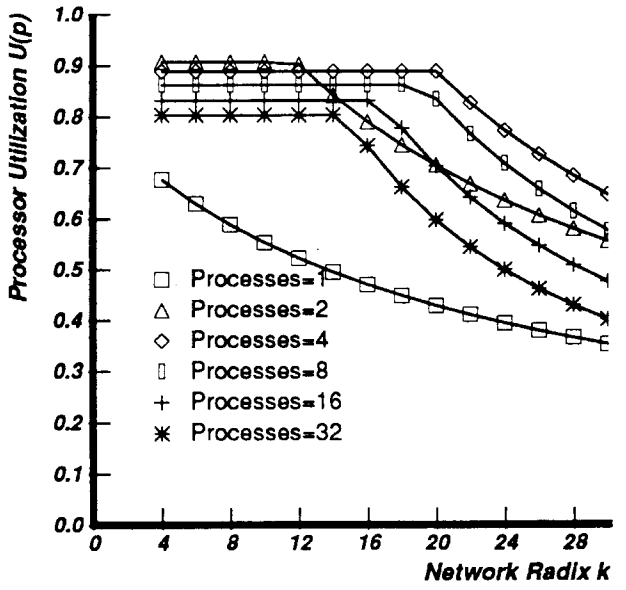

Figure 9: Impact of network and memory latency on processor utilization while considering miss rate increase due to multithreading. Varying k.

p	Overhead 1				4				16			
	Latency 18		23		18		23		18		23	
1	0.87	(.885)	0.84	(.851)	0.87	(.885)	0.84	(.851)	0.87	(.885)	0.84	(.851)
2	0.99	(.940)	0.99	(.928)	0.95	(.940)	0.95	(.928)	0.83	(.861)	0.83	(.862)
4	0.98	(.936)	0.98	(.933)	0.93	(.904)	0.93	(.901)	0.76	(.778)	0.76	(.778)

Table 6: Processor utilizations predicted by the model and measured from simulations for context-switch overheads 1, 4, and 16, memory latencies 18, and 23, and processes 1, 2, and 4.

head. The cache miss rate is modeled as the cache size per process raised to some power (both powers -1 and -0.5 are tried), and the cache size per process is chosen to be the total cache size divided by the number of processes. Our analyses shows that a similar relationship between the cache miss rate, cache size, and number of processes is valid only when the fixed miss rate is negligible compared to the interference component, and when cache sizes are smaller than the individual process working sets (see discussion following Equation 8). However, in practice, the fixed miss rate for large caches usually dominates over the interference components, and process working sets are rarely bigger than cache size.

Halstead uses a queueing model to obtain the probability that at least one process is available to run on the processor, which is simply the processor utilization. Although we assumed constant switch times and constant service times dependent on the number of processes for our results, we showed how exponential distributions for the above intervals can be used in our model in Section 3.1. In Halstead's analyses more contexts always yielded better processor utilization, while in ours, more contexts often harmed utilization.

Weber and Gupta conducted a simulation study to explore the benefits of multithreading using address traces from parallel applications [36]. In the parameter ranges of the traces they measured, our analytical results yield the same conclusions. When the network latency is fairly small (say less than 50 cycles), very few (2 to 4) contexts are sufficient to yield close to complete processor utilization. Often, when the active contexts were increased beyond a certain number, the utilization dropped due to the effects of context-switching overhead.

We conducted a validation experiment with the LocusRoute trace that was also used in Weber and Gupta's simulation study. We measured parameters from the trace, such as process working-set sizes and the constant c, and derived processor utilizations for parameter ranges considered in the simulation study. We found a good match between the model predictions and the simulation numbers. For instance, consider Table 6. Analytically computed processor utilizations for varying context-switch intervals, fixed network latencies, and processes are shown, along with the corresponding numbers from the simulation study in parentheses. The same numbers are plotted in Figure 10; the simulation graphs from [36] are on the left. We observe that the model is fairly accurate in predicting not only the relative effects of varying several parameters, but also the absolute processor utilizations.

9 Directions for Future Work

The multithreaded processor model can be extended in several ways. The benefits of multithreading in hiding synchronization delays can be studied. Indirect multistage network models such as that presented in Appendix D can be used in place of our direct network. The negative

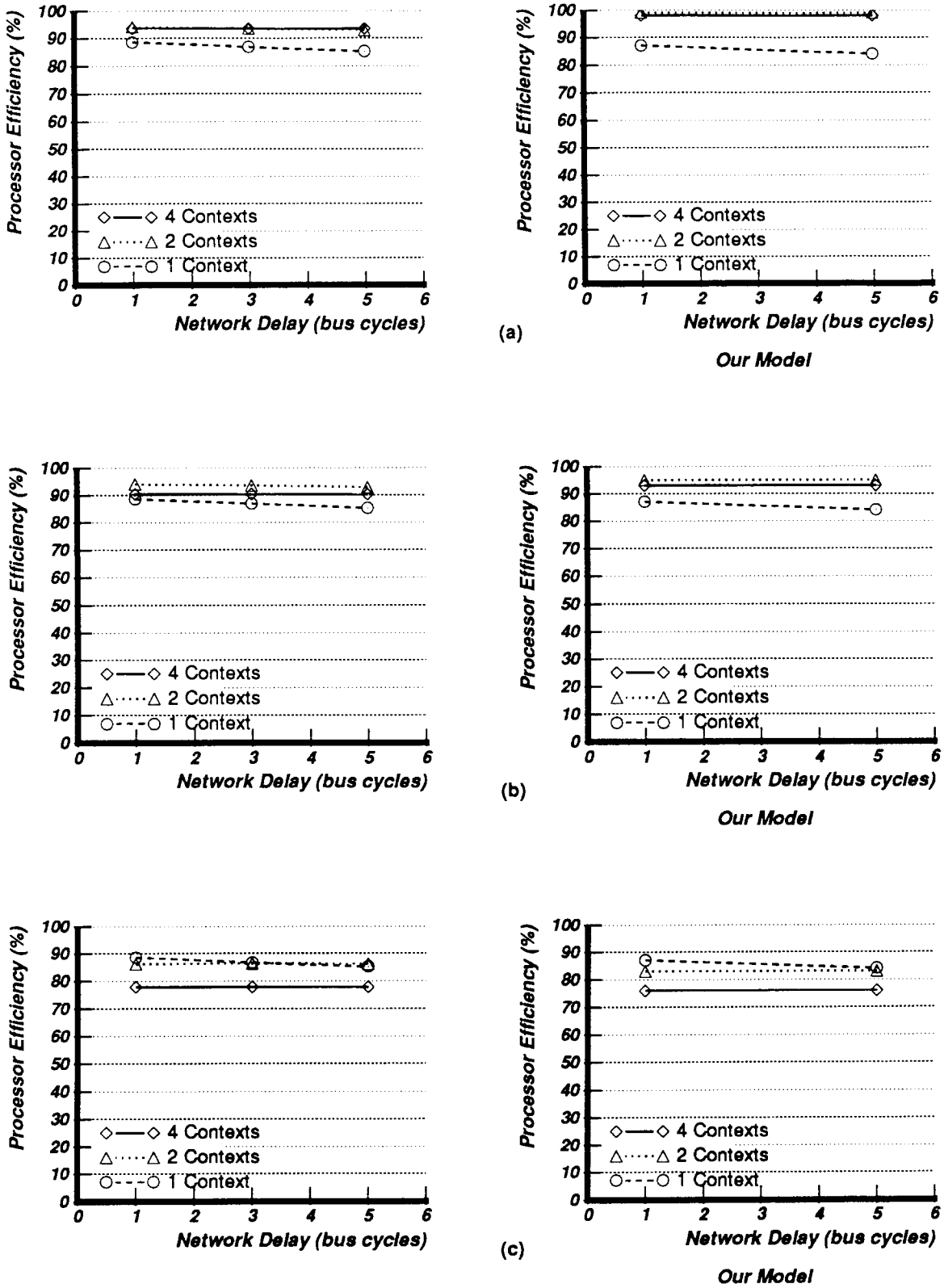

Figure 10: Comparing the model with simulations. (a) Switching overhead 1, (b) 4, and (c) 16.

effects of network contention with an increasing number of processes is smaller for these higher bandwidth networks.

An interesting area for research involves the impact of grain size and degree of parallelism on working-set sizes of threads. A reduction in the working-set size of threads is expected with increased parallelism, a trend our simulations confirm. Furthermore, with increased parallelism, working sets of the processes resident on the same processor may tend to intersect more frequently, which reduces their interference. Such a decrease implies that caches can support more than one thread without significant loss of performance.

What is a good scheduling strategy for threads? Multiple threads scheduled on a few processors will yield good processor utilizations, potential sharing of data in the cache, and communication locality, but will suffer from cache interference and poor load balancing. A performance study of these effects would be useful.

When is it best to swap process contexts from register frames into main memory and schedule other processes instead? In our current SPARCLE implementation, swapping a process into main memory takes roughly 150 cycles.

The design of a multithreaded processor must solve several new problems in cache coherence protocol design that are unique to multithreading. The possibility of systematic collisions between blocks from different threads must be addressed. For example, a thread switched out on a cache miss might find that its requested block was purged by an intervening thread soon after the block was fetched from memory. This cycle can repeat indefinitely if the two processes access blocks that map into the same cache location. A similar thrashing can occur between threads in different processors. The directory-based cache coherence protocol in Alewife has been modified to ensure forward progress in such situations [13].

Another potential problem can arise if busy-wait spinning is the mode of synchronization used. By not relinquishing the processor, a thread busy-waiting on a lock can preclude the release of the lock held by another thread awaiting its turn on the processor. Some proposals include a timeout interrupt [36]; ours forces a context switch after a given number of unsuccessful tests. In SPARCLE, synchronization is achieved using full/empty bits that can trap the processor on an access to a locked block. The cache controller can also use a trap signal whose handler unloads the thread from the processor.

10 Conclusions

Multithreading a processor to allow overlapping memory and network access with computation is a useful technique for improving processor utilization. We have developed a novel analytical model for multithreaded processors with caches. The model accounts for network contention, context-switching overhead, and data sharing. Our analysis shows that network bandwidth constraints limit the maximum achievable processor utilization, irrespective of the degree of multithreading.

Provided sufficient network bandwidth exists, multithreading is useful under a wide range of cache, processor and network parameter variations. In general, as few as two or four processes are sufficient to achieve high processor utilization. High cache miss rates due to intensive multithreading can significantly hurt processor performance. Lowering the working-set sizes of the individual threads dramatically improves processor performance. Large caches yield close to full processor utilization with as few as two to four processes, while small caches cannot achieve

high utilization with any number of processes due to increased network contention.

If portions of the working sets of the threads are shared, the negative cache effects of multithreading are mitigated, for the cost of fetching shared blocks is amortized over several threads. Multithreading is shown to be beneficial when the fixed cache miss rate is large compared to the intrinsic interference component of misses. Multiprocessor simulations of parallel applications indicate that such behavior is increasingly likely as we exploit parallelism at finer grains.

In cache-based systems, because the rate of context switches forced by remote memory operations is expected to be small compared to systems without caches, high processor utilizations (over 80%) can be achieved even if the context switch overhead is slightly large (say ten cycles). Such a relaxation of context-switch time requirements can tremendously simplify the design of multithreaded processors that have high single-thread performance.

11 Acknowledgments

The research reported here has benefited significantly from discussions with Bert Halstead, Tom Knight, Greg Papadopoulos, Juan Loaiza, Bill Dally, Steve Ward, and Randy Osborne. My gratitude to the Alewife team, including David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum, for their contributions and efforts. Jonathan Rose wrote the LocusRoute program, Larry Soule wrote the PTHOR program at Stanford, and Anoop Gupta is responsible for the parallel OPS5 implementation. Mathews Cherian and Kimming So are responsible for the SIMPLE trace. Wolf-Dietrich Weber and Anoop Gupta supplied their simulation data for the LocusRoute trace. DEC and IBM made trace data available to us. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, and by grants from the Sloan foundation and IBM.

References

[1] H. Sullivan and T. R. Bashkow. A Large Scale, Homogeneous, Fully Distributed Parallel Machine. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 105–117, March 1977.

[2] B.J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[3] W. J. Kaminsky and E. S. Davidson. Developing a Multiple-Instruction-Stream Single-Chip Processor. *Computer*, 66–78, December 1979.

[4] E. S. Davidson. A Multiple Stream Microprocessor Prototype System: AMP-1. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 9–16, IEEE, New York, May 1980.

[5] R.H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, IEEE, New York, June 1988.

[6] R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Hawaii, June 1988.

[7] G.M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* Technical Report TR-432, MIT Laboratory for Computer Science, Cambridge, MA, August 1988.

[8] Mark R. Thistle and Burton J. Smith. A Processor Architecture for Horizon. In *Proceedings of Supercomputing '88*, November 1988.

[9] Rishiyur S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[10] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[11] W. J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, IEEE, New York, June 1987.

[12] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9–24, August 1988.

[13] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. August 1990. MIT VLSI Memo. Submitted for publication.

[14] J. L. Hennessy and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[15] SPARC Architecture Manual. 1988. SUN Microsystems, Mountain View, California.

[16] M. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Thesis, Computer Science Division (EECS) UCB/CSD 83/141, University of California at Berkeley, October 1983.

[17] D. Patterson and C. Sequin. A VLSI RISC. *Computer*, 8–21, September 1982.

[18] David W. Wall. Global Register Allocation at Link Time. In *SIGPLAN '86, Conference on Compiler Construction*, June 1986.

[19] P. A. Steenkiste and J. L. Hennessy. A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[20] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[21] Leonard Kleinrock. *Queueing Systems, Volume 1.* John Wiley & Sons, 1975.

[22] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[23] Howard J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* McGraw-Hill, 1990. Second Edition.

[24] Clyde P. Kruskal and Marc Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

[25] Anant Agarwal. Limits on Network Performance. November 1989. Laboratory for Computer Science, M.I.T. MIT VLSI Memo 1989. Submitted for publication.

[26] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, May 1989.

[27] Dominique Thiebaut and Harold S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5:305–329, November 1987.

[28] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[29] Makoto Kobayashi and Myron H. MagDougall. The Stack Growth Function: Cache Line Reference Models. *IEEE Transactions on Computers*, 38(6), June 1989.

[30] Dominique Thiebaut. On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss ratio. *IEEE Transactions on Computers*, 38(7), July 1989.

[31] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[32] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[33] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, IEEE, New York, June 1988.

[34] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[35] Robert H. Halstead. Processor Architecture for Multiprocessors. 1985. Unpublished Report.

[36] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1989.

[37] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

A Extensions to the Multithreaded Cache Model

This section suggests changes to the cache model that are required to account for data sharing among processes and non-stationary behavior. We start with Equation 12 for the component of the cache miss rate due to multithreading,

$$m'(p) = v'(p)\frac{(p-1)}{S}\frac{u}{\tau} + \frac{c}{\tau}u\left(1-\frac{1}{S}\right)^u\left(1-\frac{v'(p)}{v}\right)$$

where,

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

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

and when $(p-1)u << S$,

$$m'(p) \approx \frac{cu^2}{S\tau}(p-1)\left(1+\frac{1}{c}\right)$$

A.1 Process Sharing

Portions of process working sets that are shared are immune to interference from other processes. Furthermore, the non-stationary component of the miss rate, m_{ns}, is reduced because only one process need fetch the shared blocks into the cache for the first time. Let u_{priv} be the number of blocks that are private to a given process out of the total u blocks in its working set. The multithreaded miss rate component becomes,

$$\begin{aligned} m'(p) \;=\; & v'_{priv}(p)\frac{(p-1)}{S}\frac{u_{priv}}{\tau} \\ & +\frac{c}{\tau}u\left(1-\frac{1}{S}\right)^u\left(1-\frac{v'_{priv}(p)}{v_{priv}}\right)\left(\frac{u_{priv}}{u}\right) \\ & -m_{ns}\frac{(u-u_{priv})}{u}\left(1-\frac{1}{p}\right) \end{aligned} \tag{18}$$

where,

$$v'_{priv}(p) = \frac{v_{priv}}{1+v_{priv}\frac{(p-1)}{S}}$$

$$v_{priv} = S\left[1-\left(1-\frac{1}{S}\right)^{u_{priv}}\right]$$

The first term in Equation 18 is the context-switching component $m'_{cs}(p)$ as before. Because shared cache blocks are not impacted by context switching, all the working-set related variables are replaced by the corresponding private working set and derived variables.

The derivation of the second term, however, is not as straightforward. Recall, in Equation 11, $u(1-1/S)^u$ is the number of blocks of a process that do not collide in a single-process cache, and $(1-v'(p)/v)$ is the extra fraction of blocks that become involved in collisions in the multithreaded cache. We now multiply this fraction by (u_{priv}/u) so that only the private blocks in the working set are included in this extra component.

The third term represents the decrease non-stationary component of the miss rate, and can be derived as follows. The portion of the non-stationary miss rate due to shared references for a single process can be estimated as $m_{ns}(u-u_{priv})/u$, which when divided by p yields the amortized non-stationary component of the miss rate for blocks shared between p processes. The non-stationary miss rate for blocks private to a process is $m_{ns}u_{priv}/u$, yielding an aggregate

non-stationary miss rate of $m_{ns}(u - u_{priv})/(up) + m_{ns}u_{priv}/u$. The difference between m_{ns} and this value yields an effective reduction of $m_{ns}\frac{(u-u_{priv})}{u}\left(1 - \frac{1}{p}\right)$ in the non-stationary component of the miss rate.

When $(p-1)u << S$, ignoring the non-stationary effects, we can further simplify the miss rate expression as

$$m'(p) \approx \frac{cu_{priv}^2}{S\tau}(p-1)\left(1 + \frac{1}{c}\right)$$

Because the effective working-set size is reduced to u_{priv}, sharing in the cache can significantly decrease the multithreading interference. It is also conceivable for the non-stationary reduction term to be greater than the increase in miss rate due to multithreading, and actually resulting in improved cache hit rates. A similar anomalous miss rate reduction was observed in a multiprogrammed system due to process sharing of operating system structures [31].

A.2 Non-Stationary Effects

As Appendix B shows, modeling the effect of non-stationarity in the addressing behavior of a program for multiprogrammed caches, where the time quanta are large enough to bring in the entire process working set, is straightforward. However, the effect on a finely multithreaded cache are harder to estimate.

We first estimate the effect of non-stationarity on the context-switching component of the miss rate, $m'_{cs}(p)$. If u_{ns} blocks are renewed by a process every τ, the probability that a block of a process is dead on a context switch is $(u - u_{ns})/u$. Because purging of dead blocks do not give rise to extra multithreading misses, non-stationarity reduces the multithreading component of the miss rate. Thus, if $v'_s(p)$ is the effective size of the carry-over set (or the number of blocks left behind by a process in the cache when it switches out), we compute the effect of non-stationarity on $m'_{cs}(p)$ as,

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

where

$$v'_s(p) = v'(p)\frac{u - u_{ns}}{u}$$

We estimate the impact of non-stationarity on the intrinsic interference component of the miss rate by excluding first misses of colliding blocks that are actually dead due to non-stationarity from our computation of $m'_{intr}(p)$. The number of such misses is the product of the probability a colliding block is dead and the number of colliding blocks. Estimating the probability a colliding block is dead as u_{ns}/u, we obtain the intrinsic interference miss rate due to multithreading

$$\begin{aligned} m'_{intr}(p) - m_{intr} &= \frac{cu}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right) - \frac{u}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right)\frac{u_{ns}}{u} \\ &= \frac{u}{\tau}\left(1 - \frac{1}{S}\right)^u\left(1 - \frac{v'(p)}{v}\right)\left[c - \frac{u_{ns}}{u}\right] \end{aligned} \tag{19}$$

When $(p-1)u << S$, the effective increase in the miss rate due to multithreading in the presence of non-stationarity becomes

$$m'(p) \approx \frac{cu^2}{S\tau}(p-1)\left(1 + \frac{1}{c} - \frac{2u_{ns}}{cu}\right)$$

B Comparison with a Multiprogrammed Cache

A model for multiprogrammed caches with context-switch intervals large enough to allow a process to completely replenish its working set was derived in [26]. This model predicts the cache miss as a function of the degree of multiprogramming, the associativity of the cache, the cache size, and the individual working-set sizes of the processes. For the important special case of direct-mapped caches and uniform process parameters the model can be greatly simplified. We shall compare this simplified multiprogramming model with the model for multithreading. We will also validate the simplified model in Section B.1 after modifying it to account for non-stationary behavior in the program.

Let us first simplify the multiprogramming model from [26]. The number of multiprogramming misses that process i suffers on being rescheduled in a cache with p active processes, degree of associativity D, and number of sets S, is

$$S \sum_{d=0}^{d=D} bin\left(u_i, \frac{1}{S}, d\right) \sum_{e=0}^{e=u_{i'}} MIN(d, e+d-D) bin\left(u_{i'}, \frac{1}{S}, e\right)$$

In the above expression, the term $bin\left(u_i, \frac{1}{S}, d\right)$ is the binomial distribution and represents the probability that d blocks from the working set of process i of size u_i map into one of the S cache sets. $u_{i'}$ represents the sum of the working-set sizes of the all the other processes, and $bin\left(u_{i'}, \frac{1}{S}, e\right)$ represents the corresponding probability that e blocks of the other $(p-1)$ processes map into any cache set. The number of misses divided by the the context-switching interval yields the multiprogramming miss rate $m_{cs}(p)$. In a multiprogramming situation, the context-switching interval is constrained to be large enough to allow complete replenishment of the working set. For the rest of this analysis let us assume that the context-switching interval is τ. (Recall that τ was the interval used to measure the working-set size).

If the working-set size of each process is u, we can substitute $u_i = u$, and $u_{i'} = (p-1)u$. Furthermore, for direct-mapped caches $D = 1$, and the above equation simplifies to

$$\begin{aligned} m_{cs}(p) &= \frac{S}{\tau} bin\left(u, \frac{1}{S}, 1\right)\left[1 - bin\left((p-1)u, \frac{1}{S}, 0\right)\right] \\ &= \frac{S}{\tau}\frac{u}{S}\left(1-\frac{1}{S}\right)^{u-1}\left[1-\left(1-\frac{1}{S}\right)^{(p-1)u}\right] \end{aligned} \tag{20}$$

In the above equation, $\frac{u}{S}\left(1-\frac{1}{S}\right)^{u-1}$ is the probability a set has *exactly one* block of process i mapped onto it in a direct-mapped cache,[5] and $\left[1-\left(1-\frac{1}{S}\right)^{(p-1)u}\right]$ is the probability that at least one of the blocks of the intervening $(p-1)$ processes maps into that cache set purging the block of process i. The product of the above two probabilities and S yields the number of blocks of process i purged.

[5] As explained in [26], we do not count purges of blocks from sets that have more than one block of process i mapped onto it because such blocks are more likely to be purged by the intrinsic interference within the process itself. A simpler model, albeit inaccurate, might consider all the blocks left behind in the cache by process i, yielding, $\left(1-\left(1-\frac{1}{S}\right)^{u}\right)$ for the probability that a set has *at least one* block mapped to it. Note that both expressions (using exactly one and at least one) simplify to $\frac{u}{S}$ when $u << S$, which is often the case. Simulation with address traces confirmed that the two approximations gave virtually indistinguishable results for large caches.

For $S >> pu$ the multiprogramming miss rate component further simplifies to

$$m_{cs}(p) = \frac{u^2}{S\tau}(p-1) \tag{21}$$

The above relation shows that the multiprogramming miss rate in large caches, for a low degree of multiprogramming, is linearly related to the number of processes and to the square of their individual working set sizes.

We will now contrast this miss rate with the trend in fine-grain multithreaded caches. From Equations 15 and 17 the multithreading induced miss rate

$$m'(p) = \frac{cu^2}{S\tau}(p-1)\left(1+\frac{1}{c}\right)$$

Interestingly, the miss rate for both fine and coarse grain multiprogramming is proportional to the product of the square of the working set u and the number of processes, and inversely proportional to the cache size; the difference in their performance lies in their constants of proportionality. That is,

$$\frac{m'(p)}{m_{cs}(p)} = c\left(1+\frac{1}{c}\right)$$

where typical measured values for c are between 1.5 and 2.5.

B.1 Correcting the Multiprogramming Model for Non-Stationarity

An inaccuracy with the above model was our assumption that *every* block left behind in the cache when a process relinquished the processor would be reused by the process on its return. Consequently, if the working set changed significantly between context switches the model overestimates the multiprogramming misses. This inaccuracy is easily fixed by reducing the working-set size of the process i that relinquishes the processor by the fraction of blocks that are renewed by the process. The number of blocks that are fetched during each context-switch interval for the first time can be estimated as the total number of unique blocks in a trace divided by the number of intervals [26]. Let this number be denoted as u_{ns}. Then the miss rate can be written as

$$m_{cs}(p) = \frac{S}{\tau}\left(\frac{u-u_{ns}}{S}\right)\left(1-\frac{1}{S}\right)^{u-u_{ns}-1}\left[1-\left(1-\frac{1}{S}\right)^{(p-1)u}\right] \tag{22}$$

Figure 11 compares the multiprogramming miss rate for the model and simulations with the IVEX, LocusRoute, and SIMPLE traces, for the cache parameters used in Section 6. The simulations were conducted as follows. We extracted all the references of a single process from a multiprogrammed trace and round-robin scheduled p instances of this single-process trace to simulate a multiprogramming level of p, assuming a context-switch interval of 10,000 references. Each instance of the trace was assigned a random number as a process identifier (PID). The PID of each constituent process was hashed in with the addresses to randomize the locations in the cache occupied by corresponding references of the p subprocesses. It is easy to see that the model compares well with simulations.

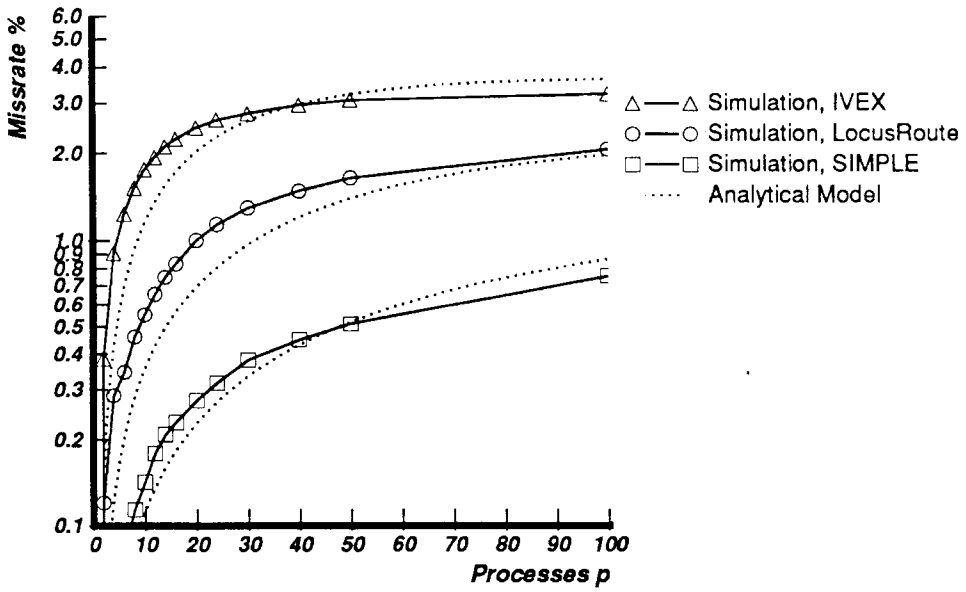

Figure 11: Increase in the miss rate due to multiprogramming.

C A Simple Direct Network Model

Our model for buffered, k-ary n-cube, direct networks proceeds along the lines of the buffered indirect network analysis of Kruskal and Snir [24]. In an n-dimensional direct network, each switch has n network inputs and n network outputs, and a port leading to the processor connected to the node. As is common in many contemporary direct networks, the network uses cut-through routing, and routes messages completely in one dimension before the next, in the order of decreasing dimension. (See [1] for such a routing scheme.) We will derive network latency as a function of the channel utilization in the network.

Let us first derive an expression for the delay in a switching node. The output queue of a switch can be treated as a queueing server, with v_i packets joining the queue during a cycle i. v_i is a random variable that can take on values ranging from 0 through n. (We use the terms packet and message equivalently). The v_i for different values of i are independent random variables; let their expectation be E and variance be V. E is simply the expected number of arrivals in any given cycle. As in [24], for such a $M/G/1$ queueing system [21], the average waiting time for a packet for a unit cycle-time system can be derived as

$$w = \frac{V}{2E(1-E)} - \frac{1}{2} \tag{23}$$

We now need the distribution of the random variable v. In an indirect network, v has a simple binomial distribution because a packet from any one of the multiple input channels is steered towards any one of the output queues with equal probability. In high-radix, direct networks that route packets completely in one dimension before the next, this is not the case. The packets have a high probability of continuing in the same dimension when the network radix is large. This situation yields a more complicated distribution for v.

Let the average number of hops taken by a packet in a dimension be k_d. Given random destinations, we can compute k_d as the expected distance between two randomly chosen points

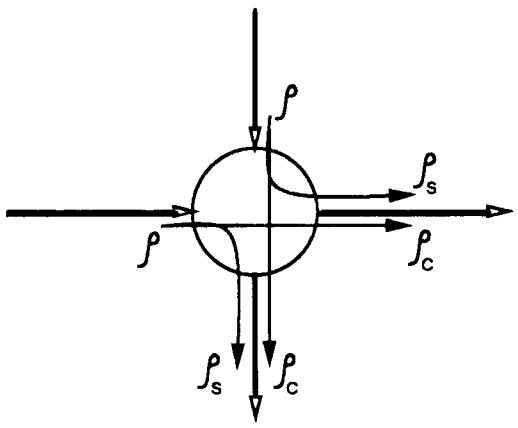

Figure 12: Channel utilizations at a switch in a direct network. ρ is the probability of a packet at an input port of the switch, ρ_c is the probability of continuing along the same dimension, and ρ_s is the probability of switching dimensions.

in a given dimension. For a network with unidirectional channels and end-around connections, $k_d = (k-1)/2$. When there are bidirectional channels, or when separate channels exist in both directions, $k_d = k/4$. When the end-around connections do not exist, $k_d = k/3$.

For a network with a high radix k, a packet at an entering channel in a switch will choose to continue along an outbound channel in the same dimension with a high probability [37]. Put another way, packets at an entering channel in a switch will tend to change dimensions with a low probability. Furthermore, when networks use the routing strategy in [1], the routing probability of an incoming packet at a channel in a given dimension is non-negligible only for the continuing channel in that dimension, and for the channel corresponding to one lower dimension. We will also ignore the contributions due to the channels connecting the switch to the processing node because the arrival probability of a packet from this channel is balanced by the probability a packet is removed from the network at this destination node. (We verified that a more detailed analysis separately treating the incoming and outgoing packets from the processing node [25] yields virtually identical graphs to those presented in Figures 3 through 9.)

In other words, our analysis ignores the contribution to contention at an output channel at a switch of all but two incoming *network* channels – one incoming channel corresponding to the same dimension as the output channel, and the channel for one higher dimension. Making the above approximation allows us to obtain a simple expression for the distribution of packets joining a queue. Note that the distribution that includes the contribution due to all the channels could also be derived along the same vein, but the analysis would be unnecessarily complicated. As depicted in Figures 14 and 13, the simple expression is quite accurate.

Let the probability of a packet at an incoming channel (or channel utilization) be ρ. Because a packet switches dimensions once every k_d hops, its probability of switching to one lower dimension is $\rho_s = \frac{\rho}{k_d}$, and its probability of continuing along the same dimension is $\rho_c = \rho\left(1 - \frac{1}{k_d}\right)$. These switching probabilities are depicted in Figure 12.

We can now write the distribution of v as:

$$p = \begin{cases} (1-\rho_s)(1-\rho_c) & \text{for } v = 0 \\ \rho_c(1-\rho_s) + \rho_s(1-\rho_c) & \text{for } v = 1 \\ \rho_c\rho_s & \text{for } v = 2 \\ 0 & \text{for } v > 2 \end{cases}$$

The expectation and variance of the number of packets joining a queue are,

$$E = \rho_c + \rho_s = \rho$$

$$V = \rho + 2\rho_c\rho_s - \rho^2$$

Substituting in Equation 23, we get the average number of delay cycles through a switch

$$w = \frac{\rho}{1-\rho}\frac{1}{k_d}\left(1 - \frac{1}{k_d}\right)$$

If the packet size (or message size) is B and the switch is time-multiplexed by a factor B, the delay becomes,

$$w = \frac{\rho B}{1-\rho}\frac{1}{k_d}\left(1 - \frac{1}{k_d}\right)$$

The channel utilization also increases and the value of ρ will reflect this increase. The average transit time T through the network can now be computed in terms of the channel utilization. Let memory latency be M, let the switches be pipelined (switches use cut-through routing), with single cycle transit time through a switch, and let the total number of network hops traversed be h. Then,

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

In the above, $h + B + M - 1$ is the minimum latency suffered by a request. Note that if the switch has a pipeline latency of s cycles, the switch delay will be s plus the contention component. (As shown in [25], separate treatment of the incoming and outgoing messages from the processing nodes adds an additional $(1 + 1/n)$ factor to the contention component.)

We validated the model through simulations against several network types and packet sizes. Figure 13 compares the network latency predicted by our model and through simulations for a network with $k = 10$ and $n = 3$ for several packet sizes. Figure 14 displays corresponding results for a two dimensional network with $k = 10$. For the validation, the network is assumed to have unidirectional channels with end-around connections ($k_d = (k-1)/2$). We can see that the above simple model predicts network latency accurately from low to moderate loads, which is expected to be the common region of network operation. At high loads, the model underestimates latency. We believe this is due to the contribution of the incoming and outgoing messages from the processing node, and the uneven loading of the network channels in various dimensions due to our routing strategy.

It is interesting to see that in high-radix direct networks (as observed by Dally [37] in unbuffered direct networks), because of the uneven probabilities of packet arrivals from various dimensions to an output switch channel, the contention is fairly low until a high network utilization. We also see that the network can be operated without significant loss of performance at higher capacities when the packet size is small.

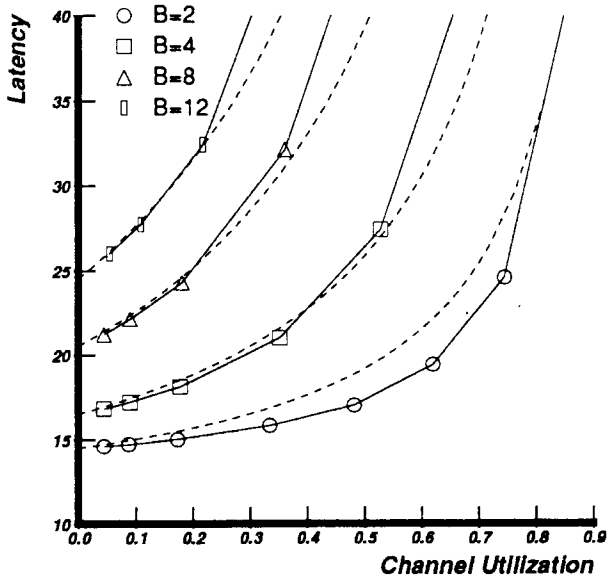

Figure 13: Comparing the direct network model with simulations. Dashed lines correspond to model predictions. $n = 3$ and $k = 10$.

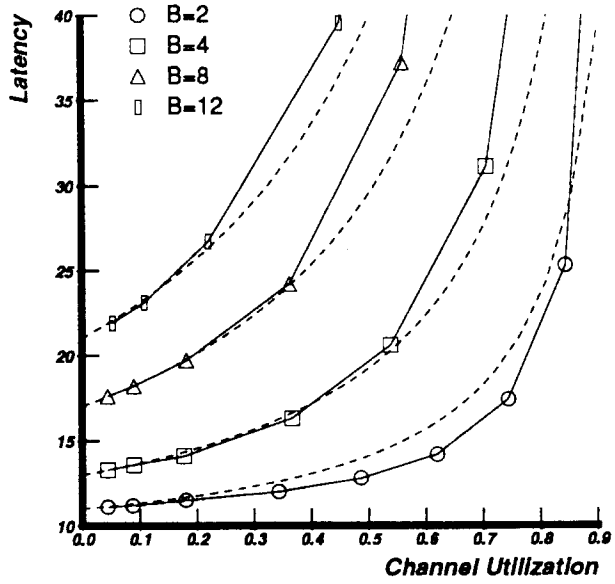

Figure 14: Comparing the direct network model with simulations. Dashed lines correspond to model predictions. $n = 2$ and $k = 10$.

D Network Latency Using an Indirect Network Model

We can also use an indirect network model in place of the direct network for analyzing the impact of multithreading. This section derives indirect network latency as a function of the number of active threads in the processor. We will use a buffered, packet-switched multistage interconnection network with $k \times k$ switches. The network model makes the usual assumptions of uniform traffic rates from all the nodes, and uniformly distributed and independent destinations. Network parameters are summarized in Table 7.

M	Memory access time
B	Message size
k	Network switch size
n	Number of network stages

Table 7: Definition of terms used in the indirect network model.

We will start with the interconnection network model proposed by Kruskal and Snir [24]. Given that the probability of a request at a network port is $p/(t+T)$, and that packets are of size B, the channel utilization can be computed as $pB/(t+T)$. The overall cache miss service time can then be computed as,

$$T = \left[1 + \frac{\frac{pB}{t+T} B \left(1 - \frac{1}{k}\right)}{2\left(1 - \frac{pB}{t+T}\right)}\right] 2n + M + B - 1 \tag{24}$$

The delay is basically the memory delay plus twice the delay through the network of n stages. The delay at each switch stage is one plus the queueing delay. This model has been used extensively in the literature. We also performed simulations with address traces of parallel applications and found that the predictions of the model were generally accurate.

Denoting the fixed network delay $T_0 = 2n + M + B - 1$, and recalling $t = \frac{1}{m}$, we can solve Equation 24 for T as,

$$T = \frac{T_0}{2} + \frac{Bp}{2} - \frac{1}{2m} + \frac{1}{2}\sqrt{\left(T_0 - pB + \frac{1}{m}\right)^2 + 4pB^2 n \left(1 - \frac{1}{k}\right)} \tag{25}$$

The chief differences between indirect network and direct network are the following: In the contention equation for the direct network model the effect of the number of threads p appears multiplied by k_d, which reflects the lower bandwidth of the direct network (given equal channel widths in the two networks). The second term under the square root is significantly lower because of the lower contention in high-radix direct networks due to the uneven probabilities of packet arrivals at an output switch channel from the incoming channels.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 501	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-87-K-0825
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (*City, State, and ZIP Code*) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (*City, State, and ZIP Code*) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

Performance Tradeoffs in Multithreaded Processors

12. PERSONAL AUTHOR(S)
Anant Agarwal

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	April 1991	39

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

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. Large caches yield close to full processor utilization with as few as two to four contexts, while small caches may require up to four times as many 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.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE (*Include Area Code*) (617) 253-5894 22c. OFFICE SYMBOL

☆U.S. Government Printing Office: 1986–507-047