Analysis of Multithreaded Architectures for Parallel Computing

Rafael H. Saavedra-Barrera
David E. Culler
Thorsten von Eicken

Computer Science Division
University of California
Berkeley, California 94720

Abstract

Multithreading has been proposed as an architectural strategy for tolerating latency in multiprocessors and, through limited empirical studies, shown to offer promise. This paper develops an analytical model of multithreaded processor behavior based on a small set of architectural and program parameters. The model gives rise to a large Markov chain, which is solved to obtain a formula for processor efficiency in terms of the number of threads per processor, the remote reference rate, the latency, and the cost of switching between threads. It is shown that a multithreaded processor exhibits three operating regimes: linear (efficiency is proportional to the number of threads), transition, and saturation (efficiency depends only on the remote reference rate and switch cost). Formulae for regime boundaries are derived. The model is embellished to reflect cache degradation due to multithreading, using an analytical model of cache behavior, demonstrating that returns diminish as the number threads becomes large. Predictions from the embellished model correlate well with published empirical measurements. Prescriptive use of the model under various scenarios indicates that multithreading is effective, but the number of useful threads per processor is fairly small.

1. Introduction

Memory latency, the time required to initiate, process, and return the result of a memory request, has always been the bane of high-performance computer architecture, and it is especially critical in large-scale multiprocessors. Many processor cycles may elapse while a request is communicated among physically remote modules, in addition to those lost during the memory access. For scalable architectures, communication latency will generally increase with the number of modules, due to the depth of the network, and may be quite unpredictable, due to network contention. Thus, the architectural challenge is to minimize the fraction of processor cycles wasted due to long-latency requests. There are two obvious alternatives: avoid latency or tolerate it. In other words, we may attempt to reduce the number of such requests or perform other useful work while requests are outstanding. In uniprocessors, the former is realized by caches [Smit82] and the latter by dynamic hazard resolution logic [Ande67, Russ78]. Unfortunately, neither of these approaches extends trivially to large-scale multiprocessors [Arvi87].

The latency avoidance strategy embodied in caches is attractive for multiprocessors because local copies of shared data are produced *on demand*. However, this replication leads to a thorny problem of maintaining a coherent image of the memory [Good83, Arga88]. Concrete solutions exist for small bus-based multiprocessors, but even these involve considerable complexity and observed miss rates on shared data are fairly high [Egge88]. Approaches that do not provide automatic replication can be expected to make a larger number of long-latency requests.

Several recent architectures adopt the alternative strategy of tolerating latency. These can be loosely classified as *multithreaded* architectures, because each processor supports several active threads of computation. While some threads are suspended, awaiting completion of a long-latency request, the processor can continue servicing the other threads. Dataflow architectures are an extreme example: the processor maintains a queue of enabled instructions and, in each cycle, removes an instruction from this queue and dispatches it into the instruction pipeline. A number of more conservative architectures have been proposed that maintain the state of several conventional instruction streams and switch among these based on some event, either every instruction [Hals88, Smit78, This88], every load instruction [Ianu88, Nikh89], or every cache miss [Webe89]. While the approach is conceptually appealing, the available evidence for its effectiveness is drawn from a small sample of programs under hypothetical implementations [Arvi88, Ianu88, Webe89]. No coherent framework for evaluating these architectures has emerged.

In this paper, we develop a simple analytical model of multithreaded architectures in order to understand the potential gains offered by the approach and its fundamental limitations. For example, what latency and remote reference rates justify multithreading, how many threads must be supported to achieve a significant improvement, and what are the trade-offs between increased tolerance and degraded cache performance due to increasing the number of threads. Our model is based on four parameters: the latency incurred on a remote reference (L), the number of threads that can be interleaved (N), the cycles lost in performing a switch (C), and the interval between switches triggered by remote references (R). L, N, and C are basic machine parameters, determined by the interconnection between modules, the amount of processor state devoted to maintaining active threads, and the implementation of the switch mechanism, respectively. R reflects a combination of program behavior and memory system design.

The basic architectural model is presented in Section 2. To develop a general understanding of the bahavior of multithreaded architectures, Section 3 analyzes the model under the assumption that the remote reference interval is fixed. A simple calculation shows that, for a given latency L, the efficiency of processor increases linearly with the number of threads up to a *saturation point* and is constant beyond that point. To provide a more realistic prediction of the processor behavior, in Section 4 we assume the remote reference interval has a geometric distribution and arrive at a Markov chain giving the efficiency as a function of N, C, L and the mean value of R. The size of this chain is extremely large, making direct numerical solutions infeasible, however, an exact solution is obtained. It gives a more accurate estimate of the processor efficiency and identifies the *transition region* between linear improvement and saturation. We can see how increasing the number of contexts enlarges the saturation region, with respect to L, and that the effects of switching overhead are most pronounced in saturation.

To make the model still more realistic, in Section 5 we embellish it to include the effects of per-processor cache degradation due to competing threads. In this case, the processor efficiency improves more slowly with the number of contexts, for a given latency, and reaches a peak, from which it may drop with further increase in N. Section 6 compares predictions of the model with published measurements obtained through trace-driven simulation, showing the correlation to be quite good. Section 7 employs the model to determine maximum improvement, net gain, and points of diminishing return under various operating assumptions in multithreaded architectures. To the extent that these predictions can be verified, this analytic approach deepens our understanding of the behavior of this new class of machines.

2. Multithreaded Processor Model

In this section, we present the architectural model and definitions used throughout the paper. We focus on one processor in a multiprocessor configuration and ignore issues of synchronization[Arvi87]. In addition, we assume the latency in processing a request is determined primarily by the machine structure and minimally affected by program behavior, so it is considered constant. We also assume that many outstanding requests can be issued and will be processed in a pipelined fashion. We say nothing about how multiple threads are actually supported by the processor and assume only that there are N of them per processor and it takes a constant time C to switch from one to another. A thread is represented by a *context*, consisting of a program counter, a register set, and whatever context status words. The model is optimistic, in that it assumes sufficient parallelism exists throughout the computation.

A conventional single-threaded processor will *wait* during a remote reference, so we may say it is idle for a period of time L. A multithreaded processor, on the other hand, will suspend the current context and switch to another, so after some fixed number of cycles it will again be busy doing useful work, even though the remote reference is outstanding. Only if all the contexts are suspended (*blocked*), will the processor be idle. Clearly, the objective is to maximize the fraction of time that the processor is busy, so we will use the efficiency of the processor as our performance index, given by

$$\varepsilon = \frac{Busy}{Busy + Switching + Idle}, \tag{1}$$

where $Busy$, $Switching$, and $Idle$ represent the amount of time, measured over some large interval, that the processor is in the corresponding state. The basic idea behind a multithreaded machine is to interleave the execution of several contexts in order to dramatically reduce the value of $Idle$, but without overly increasing the magnitude of $Switching$.

The state of a processor is determined by the disposition of the various contexts on the processor. During its lifetime, a context cycles through the following states: ready, running, leaving, and blocked. There can be at most one context running or leaving. A processor is *busy*, if there is a context in the running state; it is *switching* while making the transition from one context to another, i.e., when a context is leaving. Otherwise, all contexts are blocked and we say the processor is *idle*. A running context keeps the processor busy until it issues an operation that requires a context switch. The context then spends C cycles in the *leaving* state, then goes into the *blocked* state for L cycles, and finally re-enters the *ready* state. Eventually the processor will choose it and the cycle will start again.

This general model can be applied to several different multithreaded architectures, with varying switching policies. We briefly present three examples and give an interpretation of the run length and memory latency for each. First, suppose a context is preempted when it causes a cache miss. In this case, R is taken to be the average interval between misses (in cycles) and L the time to satisfy the miss. Here, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles. A more conservative approach is to switch on every load, independent of whether it will cause a miss or not. In this case, R represents the average interval between loads. Our general multithreading model assumes that a context is blocked for L cycles after every switch, but in the case of a "switch on load" processor this only happens if the load causes a cache miss. The general model can be employed, if we postulate that there are two sources of latency (L_1 and L_2) each having a particular probability (p_1 and p_2) of occurring on every switch. If L_1 represents the latency on a cache miss, then p_1 corresponds to what is normally referred as the miss ratio. L_2 is a zero cycle memory latency with probability p_2. Finally, consider a processor that switches on every instruction, independent of whether it is a load or not ("switch always"). As in the case of the "switch on load" processor, we create two sources of latency, L_1, representing a cache miss latency, with p_1 being the probability of a cache miss per instruction. The analysis in this paper is in terms of the general model, however, some of the examples assume a "switch on miss" processor.

3. Concepts and Preliminary Analysis

In order to introduce the basic concepts and terminology, we first assume the context transition times, R and C, are constant. In later sections, we consider more realistic conditions. A single-threaded processor executes a context until a remote reference is issued (R cycles) and then is idle until the reference completes (L cycles), before continuing. There is no context switch and, obviously, no switch overhead. We can model this behavior as an alternating renewal process having a cycle of $R + L$. In terms of eq. (1), R and L correspond to the amount of time during a cycle that the processor is $Busy$ and $Idle$, respectively. Thus, the efficiency of the single-threaded machine is given by

$$\varepsilon_1 = \frac{R}{R+L} = \frac{1}{1+L/R}. \tag{2}$$

This shows clearly the performance degradation of such a processor in a parallel system with large memory latency.

With multiple contexts, memory latency can be hidden by switching to a new context, but we assume the switch takes C cycles of overhead. Assuming the run length between switches is constant, with a sufficient number of contexts there is always a context ready to execute when a switch occurs, so the processor is never idle. In this case, we say the processor is *saturated*. The cycle of the renewal process in this case is $R + C$, and the efficiency is simply

$$\varepsilon_{sat} = \frac{R}{R + C} = \frac{1}{1 + C/R}. \tag{3}$$

Observe, that the efficiency in saturation is independent of the latency and also does not change with a further increase in the number of contexts. Saturation is achieved when the time the processor spends servicing the other threads exceeds the time to process a request, i.e., when $(N - 1)(R + C) > L$. This gives the saturation point, under constant run length, as

$$N_d = \frac{L}{R + C} + 1. \tag{4}$$

When the number of contexts is below the saturation point, there may be no ready contexts after a context switch, so the processor will experience idle cycles. The time to switch to a ready context, execute it until a remote reference is issued, and process the reference is equal to $R + C + L$. Assuming N is below the saturation point, during this time all the other contexts have a turn in the processor. Thus, the efficiency is given by

$$\varepsilon_{lin} = \frac{NR}{R + C + L}.$$

Observe, the efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant. The equation for ε_{sat} gives the fundamental limit on the efficiency of a multithreaded processor and underlines the importance of the ratio C/R. Unless the context switch is extremely cheap, the remote reference rate must be kept low.

4. Analysis with Stochastic Run Lengths

In this section, we improve the basic model by assuming the run length of a context (R) is a random variable having a geometric distribution; i.e., the probability of executing an instruction that will trigger a context switch is $p = 1/R$. We present an exact solution to the Markov chain and compare it to the solution of the simple model above.

The behavior of a multithreaded processor is conveniently represented by a Petri net diagram as in Figure 1. The circles represent *places*, and the boxes *transitions*. Four of the five places correspond directly to the four context states; place A is used to enforce the constraint that only one context can be running or leaving. Note that transition E is immediate, while transitions C and L are deterministic (black rectangle). The stochastic transition R associated with the running state of a context is shown as a white rectangle.

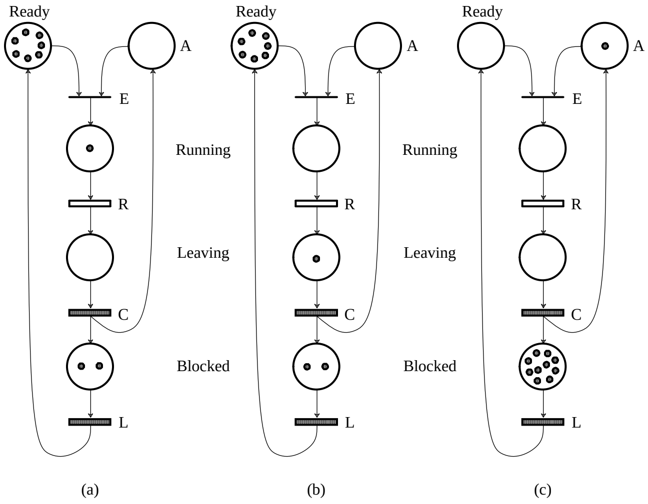

Figure 1: A Petri net representing the three states of the multithreaded processor. In (a) the processor is *busy*; in (b) the processor is *switching*; and in (c) the processor is *idle*.

4.1. An Exact Solution to the Markov Chain

By computing the reachability set of a Petri net we obtain the Markov chain, which can be solved numerically to obtain the efficiency for specific values of N, R, C, and L. However, it is computationally unfeasible to obtain a solution when the number of contexts is not small; the number of states in the reachability set is too large. The exact number of states is given by

$$States = 1 + L + C + \sum_{i=2}^{N} \binom{L - (i-2)C}{i}.$$

For example, with a latency of 128 cycles, a context switch time of 2 cycles, and 4 contexts, the Markov chain has more than 9 million states. Fortunately, we have found the exact solution to the limiting probabilities. The main steps in the derivation are sketched below; details can be found in [Saav90].

Although the number of equations in the transition matrix for the Markov chain is very large, there are several patterns present in this particular system that makes it possible to reduce all equations to a new system having only one unknown. Let the limiting probability for the this state be π_1. We group all the limiting probabilities that represent the same processor state together ($Busy$, $Switching$, or $Idle$) and arbitrarily set $\pi_1 = 1$. We then get the following particular solution to the set of equations given by the transition matrix

$$\Pi_{Busy} = 1 + \sum_{k=1}^{N-1} \sum_{j=1}^{C} \binom{L - (k-1)C - j}{k-1} \frac{p^k}{(1-p)^{(C+1)(k-1)+j}} + \sum_{k=1}^{N-1} \binom{L - kC}{k} \frac{p^k}{(1-p)^{(C+1)k}}, \tag{5}$$

$$\begin{aligned}\Pi_{Switching} = {} & \sum_{j=1}^{C} \frac{p}{(1-p)^j} + C \binom{L-(N-1)C}{N-1} \frac{p^N}{(1-p)^{(C+1)(N-1)}} + \\ & \sum_{k=1}^{N-2} \sum_{j=1}^{C} \binom{L - kC - j}{k} \frac{p^{k+1}}{(1-p)^{(C+1)k+j}} + \\ & + \sum_{j=1}^{C} \sum_{i=1}^{C-j} \binom{L - (N-2)C - i - j}{(N-2)} \frac{p^N}{(1-p)^{(C+1)(N-2)+i+j}} + \\ & \sum_{k=1}^{N-2} \sum_{j=1}^{C} \sum_{i=1}^{C} \binom{L - (k-1)C - i - j}{k-1} \frac{p^{k+1}}{(1-p)^{(C+1)k+i+j}},\end{aligned} \tag{6}$$

$$\Pi_{Idle} = \binom{L - (N-1)C}{N} \frac{p^N}{(1-p)^{(C+1)(N-1)}}. \tag{7}$$

By replacing eqs. (5)-(7) into (1) we effectively eliminate π_1.

This gives us the following expression for the efficiency

$$\varepsilon = \frac{\Pi_{Busy}}{\Pi_{Busy} + \Pi_{Switching} + \Pi_{Idle}}.$$

These equations are only valid when $L > C$, but is not a serious restriction, since it does not make sense to build a multithreaded machine that takes more time to switch contexts than the memory latency it attempts to hide[1]. Using eqs. (5)-(7), it is straightforward to compute the efficiency for complex systems with many contexts, large run lengths and long memory latencies.

4.2. A Better Approximation to the Saturation Point

It is easy to see that eq. (4) is a first-order approximation to the saturation point in the case of stochastic run lengths; it only considers the average run length of a context. However, as shown by figure 2, the efficiency we obtain using N_d may be well below that in saturation. We can get a much better approximation using information about the distribution of the run lengths. Essentially, we must determine the number of contexts required to reduce the probability that the processor is idle below some threshold. Knowing that a geometric random variable with mean p has variance $(1-p)/p^2 = R(R-1)$) and making some simplifications [Saav90], we obtain a quadratic equation for the new approximate saturation point (N_s). The solution is

$$N_s = \left(\frac{\alpha R + ((\alpha R)^2 + 4(R+C)L)^{1/2}}{2(R+C)} \right)^2 + 1. \tag{8}$$

Variable α gives the level of the 'confidence' (in the sense of the number of standard deviations considered), that the processor will not enter the idle state.

Equation (8) was obtained by attempting to minimize the probability of entering the *Idle* state. However, what really interests us is the fraction of the maximum efficiency that N_s contexts provide. We obtain this by using (8) in the formulae for the efficiency given by eqs (5)-(7) and comparing it to the maximum efficiency given by (3). Substituting the expression for N_s with α = 1, the value obtained for the efficiency is more than 95% the efficiency in saturation; if α is increased to 1.5, this efficiency is more than 99% of that in saturation. When α = 0, (8) reduces to (4).

4.3. Comparing the Approximate and Exact Solutions

Intuitively, we expect the solution of the deterministic model to match well with the stochastic solution far from the transition region with a maximum separation in the transition itself. In figure 2, we plot both solutions as a function of N. The latency is 128 cycles; the probability of context switching .0625 ($R = 16$); and the context switch time 2 cycles. In this case, the maximum efficiency of the processor is .888. The saturation point (N_d) predicted by the deterministic model is located at 8 contexts. This corresponds to a real efficiency of .743 which is only 84% of the actual maximum. In contrast, N_s with $\alpha = 1.5$ equals 13 contexts and its efficiency corresponds to 99% of the maximum.

[1] However, in a "switch on load" or "switch always" processor condition $L > C$ may not be true. In these processors L corresponds to the weighted average of the multiple sources of latency, and even when the remote latency is large the effective latency we obtain can be smaller than a context switch time.

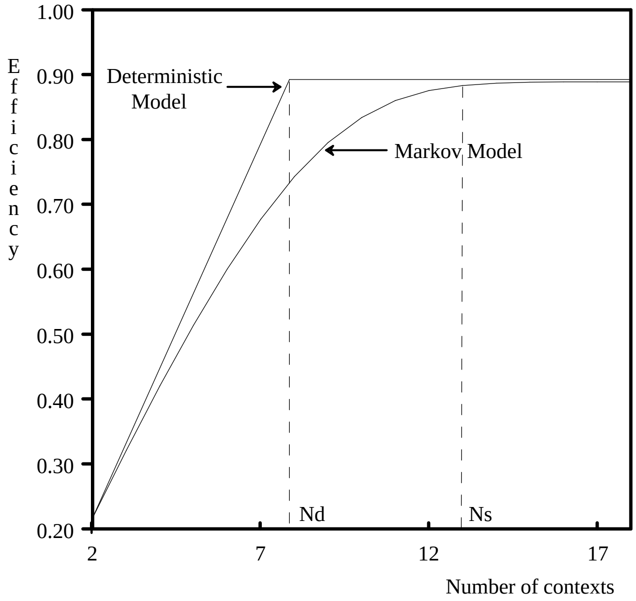

Figure 2: Efficiency as a function of the number of contexts: L = 128 cycles, R = 16 cycles, C = 2 cycles. The rightmost curve is obtained from the exact solution to the Markov chain.

4.4. Latency, Number of Contexts, and Context Switch Overhead

The utilization expression obtained with the stochastic model allows a detailed investigation of the behavior of a multithreaded processor as a function of N, R, L, and C. This is illustrated in figure 3, where we show utilization curves for a system in which memory latency is varied from zero to 200 cycles, context switch overhead is either 0, 1, 4, or 16 cycles, and the number of contexts is 2 and 6 contexts. However, the qualitative behavior we identified in the simple deterministic model is present here. (1) The maximum utilization is completely determined by the ratio C/R; (2) In the saturation region, latency is effectively eliminated; (3) The effect of the context switch overhead is greater in saturation than in the linear regime. (4) Outside the saturation region, an increase in memory latency rapidly reduces the utilization of the processor.

5. Dependency of Contexts on Cache Miss Ratio

The next step in our analysis is to consider a more realistic model that takes into account that increasing the number of contexts running on a processor will have an adverse effect on the cache miss ratio. A higher number of cache misses will decrease the run length and possibly increase memory latency due to higher memory contention. In this section, we modify the efficiency equations obtained in both the deterministic and stochastic cases to take into account the possible effect of multiple contexts on the run length. We will assume that each context uses $1/N$ of the total cache, instead of sharing the whole cache between the contexts. Thus, the total cache size is constant and, as the number of contexts increases, each context will have a smaller cache for its own use. This is a 'middle ground' assumption. In reality, contexts may interfere constructively or destructively on shared data. If they do not interfere at all, they might fit in the cache with unequal partitions.

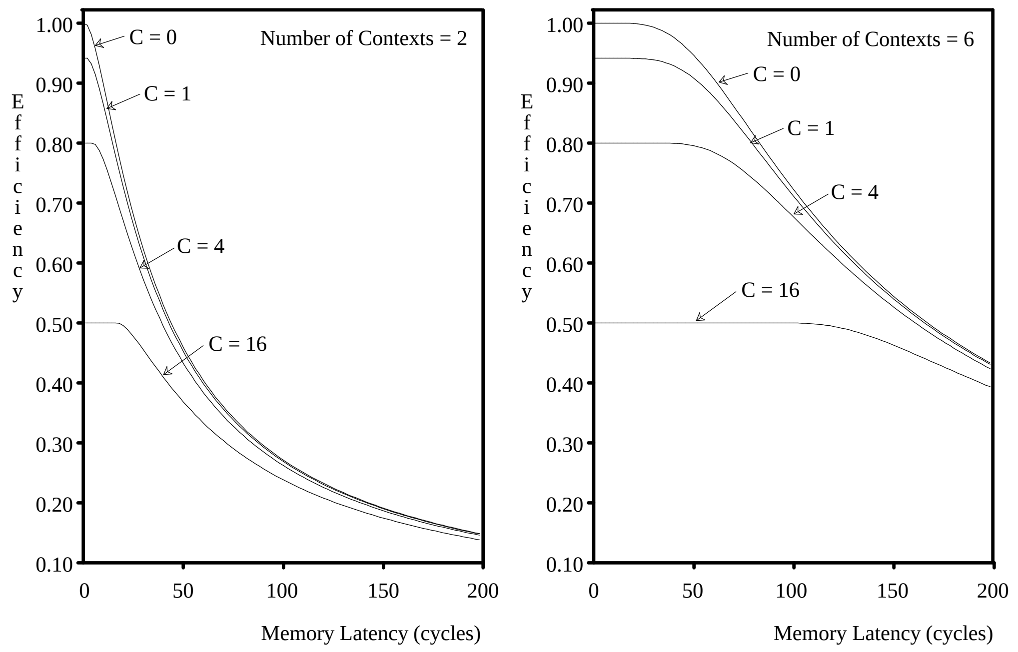

Figure 3: Efficiency as a function of latency and context switch. For both graphs $R = 16$ cycles.

On uniprocessors cache misses experienced by an application can be classify into four groups: *start-up effects*, *nonstationary behavior*, *intrinsic interference*, and *extrinsic interference* [Agar89]. The first is caused by initial execution of the program; the second by changes in the program's working set; the third by the size of the cache; and the last one by multiprogramming. Only the number of misses due to interference (intrinsic and extrinsic) increases with a larger number of contexts. By assigning independent caches to the contexts we eliminate the effect of extrinsic interferences in the miss ratio. Hence, we only have to consider how the intrinsic interference is affected[2].

Many different techniques have been used to obtain the performance of caches: trace-driven simulation, hardware measurement, and analytical models [Smit82, Clar83, Agar89]. It has been observed, although not completely validated, that the miss ratio (m), as a function of the size of the cache, can be approximated by $m = A\, S^{-K}$, where S is the cache size, and A and K are positive constants that depend on the workload[3]. The above miss ratio formula appears to be valid for caches between 1K and 256K bytes under uniprocessor execution. We will use this miss ratio approximation to obtain an expression for the miss rate rate for N contexts in terms of the miss ratio of one context using all the cache. For, example, if the total cache size is 256K bytes the above formula can be used to obtain approximate miss ratios for up to 128 contexts, each using a cache partition of 2K bytes, or 4 contexts each using a cache partition of 64K bytes.

Let $m(N)$ be the miss ratio when there are N contexts, each using a cache of size $S_N = S_1/N$. An expression for $m(N)$ can be obtained in the following way

$$m(N) = A\, S_N^{-K} = A\, S_1^{-K} \left(\frac{S_N}{S_1} \right)^{-K} = m(1)\, N^K .$$

This expression for $m(N)$ is a monotonicly increasing function and does not take into consideration that the miss ratio cannot be greater than one. Thus, a better expression for the miss ratio should be:

$$m(N) = \begin{cases} m(1)\, N^K, & \text{if } N \le \left\lfloor m(1)^{-1/K} \right\rfloor ; \\ 1, & \text{if } N > \left\lfloor m(1)^{-1/K} \right\rfloor . \end{cases}$$

Now, $m(1)$ equals p over the average number of cache queries per instruction (p_{load}), thus we can easily express the run length with N contexts as

$$R(N) = \begin{cases} R(1)\, N^{-K}, & \text{if } N \le \left\lfloor R(1)^{1/K} \right\rfloor ; \\ \dfrac{1}{p_{load}}, & \text{if } N > \left\lfloor R(1)^{1/K} \right\rfloor . \end{cases} \qquad (9)$$

The smallest possible value for $R(N)$ is the average run length between loads ($1/p_{load}$); this is the case when every load misses on the cache. An estimate of $R(1)$ and K can be obtained from uniprocessor simulations under different cache sizes.

[2] This does not reflects that there may be some minimum miss rate, regardless of cache size, due to communication of data between processors, but could be easily extended to do so.

[3] D. Thiebaut argues in [Thie89] that this relationship can be explained by considering the program's execution as a fractal random walk over the address space. In his model constant K is related to the fractal's dimension and this value depends on the particular intermiss gap distribution of the program.

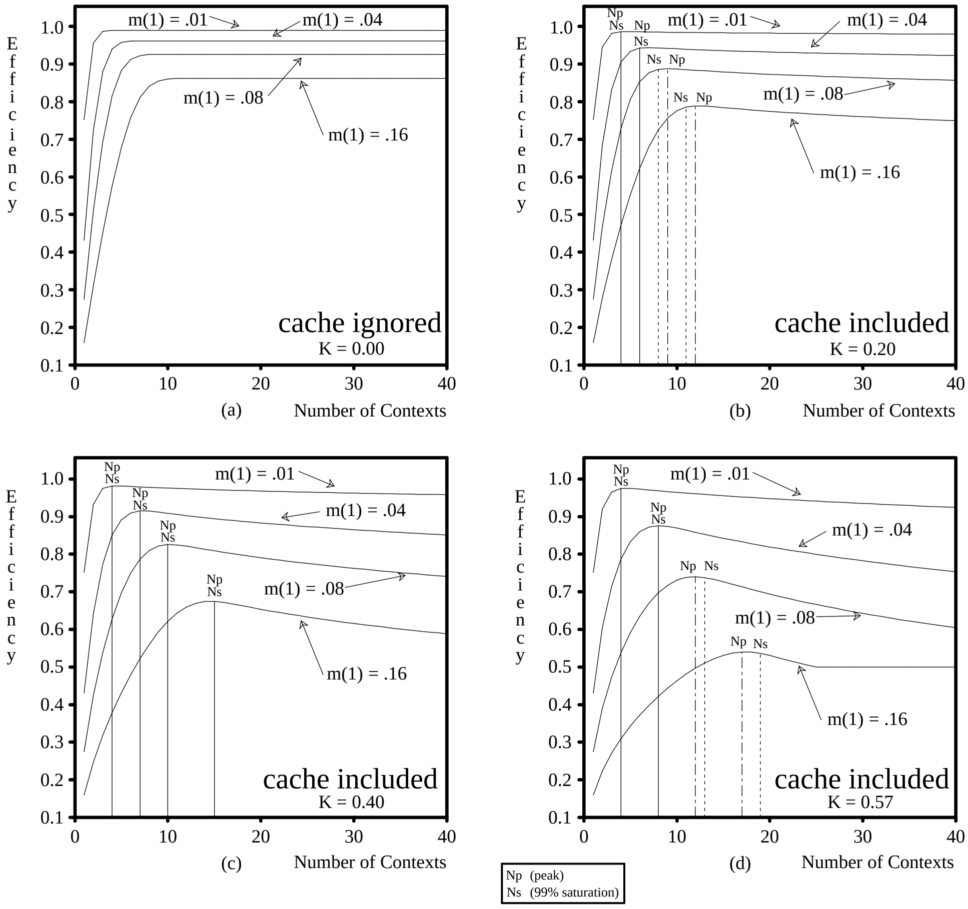

Figure 4: Efficiency versus the number of contexts for different values of $m(1)$ and K.

5.1. Results Including Cache Dependency

Equation (9) can be replaced in formulas (2)-(8) to obtain a new set of equations that reflect how the miss ratio depends on the number of contexts. Figure 4 compares the processor efficiency computed using the original equations and the new ones for various values of K. The four curves in each graph assume a single context miss ratio of 0.01, 0.04, 0.08, and 0.16, a latency nof 128 cycles, and a context switch cost of 4 cycles. The set of values for the cache degradation constant, K, are 0.0, 0.2, 0.4, and 0.57, and the size of the cache is, in the case of a single context, 256K bytes[4]. We also assume that p_{load}, the fraction of loads executed in the workload, equals 25 percent.

The graphs in figure 4 indicate that utilization degrades considerably when the initial miss ratio or the cache degradation constant is large. When $K > 0.0$, we observe that the utilization increases up to some maximum (ε_{peak}) and then drops as N increases. This maximum value diminishes as K becomes larger and more contexts are needed in order to reach the maximum. The flat segment in the curve with $m(1) = .16$ and $K = .57$ reflects a regime where every query to the cache misses, ($m(N) = 1$).

An estimate of the saturation point (N' sub s) in the presence of cache degradation is easily obtained using (8) with R replaced by R(N), as given in (9), by computing the fix-point of the resulting non-linear equation. (R(N) is monotonically nonincreasing). As demonstrated in figure 4, N'sub s, computed with lpha = 1.5, is a good predictor of the number of contexts needed to obtain maximum efficiency (Npeak). Moreover, in the deterministic model with cache degradation, it can be proved that the point where the system saturates corresponds exactly to that of maximum efficiency, independent of the values of $m(1)$ and K.

[4] For most programs K lies between .2 and .5, but is not unusual to find programs with a value as large as the one we have selected. The large value reflects a scenario where the miss ratio increases rapidly as the cache size is decreased.

6. Comparison with Published Measurements

The analytic model developed above employs only a crude characterization of program behavior and the machine organization, so the "proof of the pudding" is the ability to predict actual behavior. We now compare predictions from our model against trace-driven simulation results for a multiprocessor currently being studied at Stanford [Webe89]. The simulated architecture consists of several nodes connected via a network.

program	run length $R(1)$	latency L	cache degradation K	miss rate $m(1)$	loads per instruction
LOCUS_ROUTE	156	22	.5747	.0055	.166
P_THOR	50	25	.1988	.0160	.250
MP3D	16	32	.1315	.0350	.786

program	switch	1 context		2 contexts		4 contexts	
	cost	Weber	Model	Weber	Model	Weber	Model
LOCUS_ROUTE	1	0.890	0.876	0.946	0.975	0.946	0.986
	4	0.890	0.876	0.948	0.952	0.904	0.946
	16	0.890	0.876	0.864	0.867	0.780	0.815
P_THOR	1	0.690	0.666	0.846	0.894	0.882	0.972
	4	0.690	0.666	0.824	0.856	0.834	0.904
	16	0.690	0.666	0.728	0.723	0.660	0.703
MP3D	1	0.385	0.333	0.550	0.561	0.652	0.837
	4	0.385	0.333	0.536	0.527	0.612	0.744
	16	0.385	0.333	0.470	0.421	0.410	0.455

Table 1: Comparison between trace-driven simulation and the stochastic model modified to include the cache degradation due to the number of contexts. The first table gives the relevant parameters needed for our equations.

Each node consists of a processor supporting some number of contexts, a direct-mapped instruction and data cache of size 64K bytes, a fraction of the global memory, and a network interface. The caches are maintained consistent by a directory-based cache coherency protocol. A context executes until it requires information not available in its processor cache or attempts to write data marked "read-shared".

Restating measurements presented in [Webe89] in terms of our model we obtain values for the relevant parameters shown in the top portion of table 1. These are used as input for our model and the comparison with reported simulation results is presented in the remainder of table 1.

The predictions are within 10% of the reported simulation results, except on MP3D, for 4 contexts and a small context switch overhead. Observe that where a degradation is observed in going from 2 contexts to 4 contexts, the prediction shows a similar quality and magnitude. This behavior matches well with the efficiency curves presented figure 4. The discrepancy in the case of program MP3D seems to be a result of using the same latency for different number of contexts. Weber and Gupta report that, in the case of MP3D, an increase in the number of contexts resulted in significantly larger network delay due to higher global traffic. Unfortunately, they do not report the new value for the latency. Using our model, we calculate that the reported efficiency would be obtained with an average latency of 55 cycles, rather than 32.

7. Designing a Multithreaded Machine

We now proceed to show how our analytical model can be used to explore the space of possible design choices in building an effective multithreaded machine. To make the exercise more realistic, we will consider the parameters values used in the last section.

We consider memory latencies of 32, 64, and 128 cycles. A latency of 128 cycles may seem large, but it will help us evaluate how much multithreading can hide latency and improve efficiency. With respect to context switching, we consider four possible design points; the most aggressive design attempts to switch contexts immediately, for a value of $C = 0$; three more conservative designs require 1, 4 and 16 cycles of overhead to switch, respectively. Although a context switch overhead of 16 cycles may appear large, such a system needs only to maintain one context on-chip and keep the others off-chip in fast memory (engineering such a system is still a challenge). This implementation is interesting because it supports a large number of contexts with only a small increase in processor cost. We consider three miss ratio values of approximately 1.1%, 2.2%, and 3.1% with 1.4 memory references per instruction to give average run lengths of 16, 32 and 64 cycles. We further assume a cache degradation constant (K) of .57, which is large compared to magnitudes normally found in other workloads.

For these parameters, table 2 gives the number of contexts (N_{peak}) needed to achieve maximum efficiency and the ratio between the maximum utilization and the utilization with only one context ($\varepsilon_{peak}/\varepsilon_1$). The results show large variability in the number of contexts needed to achieve saturation and relative improvements that vary from a mere four percent to a nine fold increase.

In the real world, a machine designer never has an unlimited amount of resources; generally, the goal is to produce the largest improvement with the smallest increase in cost and complexity. Therefore, we might consider how many contexts are needed to improve utilization by some constant factor across a range of design values. As a possible answer to this question, the 'Improvement' column of Table 2 shows the number of contexts needed to increase utilization by 50 and 100 percent with respect to a single context. The results show that only three contexts are needed to achieve a 50 percent improvement, wherever this is possible. That only three contexts are sufficient to obtain a 50 percent improvement across a wide range of values for L, C, and R is not surprising, considering that with three contexts most of the configurations are in the linear regime. The improvement in utilization can be easily approximated using equation (4) as $N \cdot R(N)/R(1) \approx 1.6$. In the case of 100 percent improvement, a similar pattern emerges; only 5 contexts are required.

The last two columns of table 2 give the number of contexts needed to achieve fifty and seventy-five percent of the maximum utilization. Entries having a value of one indicate that multithreading does not provide a substantial improvement. The configurations where multithreading looks most promising are those where the last four columns have increasing values;

Parameters			Maximum		Improvement		Fraction of Maximum	
L	C	R	N_{peak}	$\varepsilon_{peak}/\varepsilon_1$	$1.5 \cdot \varepsilon_1$	$2.0 \cdot \varepsilon_1$	$.50 \cdot \varepsilon_{peak}$	$.75 \cdot \varepsilon_{peak}$
32	0	16	16	3.00	2	4	2	6
32	0	32	8	2.00	3	8	1	3
32	0	64	5	1.50	5	–	1	2
32	1	16	10	2.41	3	5	2	4
32	1	32	7	1.82	3	–	1	2
32	1	64	5	1.44	5	–	1	1
32	4	16	5	1.78	3	–	1	2
32	4	32	5	1.52	5	–	1	1
32	4	64	4	1.30	–	–	1	1
32	16	16	3	1.04	–	–	1	1
32	16	32	2	1.06	–	–	1	1
32	16	64	2	1.06	–	–	1	1
64	0	16	40	5.00	3	4	7	17
64	0	32	17	3.00	3	4	3	6
64	0	64	8	2.00	3	8	1	3
64	1	16	21	3.63	3	5	4	9
64	1	32	13	2.62	3	6	2	4
64	1	64	8	1.89	3	–	1	2
64	4	16	10	2.54	3	5	2	4
64	4	32	8	2.11	3	6	1	3
64	4	64	6	1.69	3	–	1	2
64	16	16	4	1.50	4	–	1	1
64	16	32	4	1.42	–	–	1	1
64	16	64	3	1.31	–	–	1	1
128	0	16	128	9.00	3	5	27	58
128	0	32	45	5.00	3	5	7	17
128	0	64	18	3.00	3	4	3	6
128	1	16	48	5.60	3	5	10	22
128	1	32	30	4.07	3	5	5	11
128	1	64	16	2.77	3	5	2	5
128	4	16	21	3.65	3	5	4	10
128	4	32	15	3.08	3	5	3	6
128	4	64	10	2.39	3	5	2	4
128	16	16	7	2.12	3	6	1	4
128	16	32	6	1.98	3	6	1	3
128	16	64	5	1.77	3	–	1	2

Table 2: The two columns under label 'Maximum' give the number of contexts and utilization increase with respect to a single context processor. The columns under label 'Improvement' give the number of contexts needed to improve utilization with respect to a single context by 50 and 100 percent. The last two columns give the number of contexts needed to achieve 50 and 75 percent of the maximum utilization.

meaning that it is possible to achieve good improvement in utilization and that there is room for improvement by increasing the number of contexts.

8. Conclusion

We have presented and analyzed a series of models for a multithreaded processor, each progressively more realistic than the previous one, and in each case derived efficiency equations in terms of the memory latency, context switch overhead, run length, and number of contexts. The simplest model assumes that all state transitions take a constant amount of time and ignores the effect of multithreading on the cache miss ratio. Next, we took into consideration the stochastic nature of the execution run length and found the exact solution for the corresponding Markov chain.

In the first two models the cache miss ratio was assumed to be independent of the number of contexts supported by the processor. This unrealistic assumptions was later dropped, by incorporating a new expression for the run length in terms of the number of contexts into the solution to the Markov chain. This more realistic model correlates well with the limited empirical results available to date, and the discrepencies show where further improvements in the model are possible.

Finally, we used our solution to explore the design space of a particular multithreaded machine. We showed how our equations can be used to compare various alternatives and the relative improvement in efficiency between a multithreaded processor and a conventional single context processor.

Apart from presenting and analyzing the different models, we identified the principal factors that need to be taken into account in the design of a multithreaded machine. Some of our conclusions can be summarized as follows:

- The most critical design parameter in a multithreaded architecture is the ratio C/R. Unless the context switch cost is essentially zero, the run length must be maximized, in which case the number contexts required is small.

- Within the saturation region, the utilization of the processor is only a function of the run length and the context switch overhead and substantial increases in latency can be tolerated.
- Caches are important in a multithreaded environment, as they allow large run lengths between misses and may perform well even when the number of contexts is increased. The reduction in the run lengths in a multithreaded machine due to a larger number of contexts can be modeled as reducing the size of the cache that each context can use.
- A processor supporting a small number of contexts, between 2 and 5, can expect an improvement in utilization between 50 and 75 percent over a large range of latencies, context switch overheads, and run lengths.

In this paper, we have ignored the issue of synchronization. At this point there is no general agreement in how much it will impact execution, nor of the level at which it should be handled. Whether a multithreaded machine will provide an acceptable solution to the synchronization problem remains an interesting problem that will require considerable attention in the near future.

Acknowledgements

We would to thank Jose A. Ambros-Ingerson, Herve Touati, Bob Boothe and David Cross for their comments on earlier drafts. The material presented here is based on research supported in part by NASA under consortium agreement NCA2-128 and cooperative agreement NCC2-550. Computing resources were provided in part by the National Science Foundation through the UCB Mammoth project under grant CDA-8722788.

References

[Ande67] Anderson, D.W., Sparacio, F.J., and Tomasulo, R.M., ‘‘The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling’’. *IBM Journal*, Vol.9, No.25, January 1967, pp. 8-24.

[Arga88] Agarwal, A., Simoni, R. Hennessy, J., and Horowitz, M., ‘‘An Evaluation of Directory Schemes for Cache Coherency’’. *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1988, pp. 280-289.

[Arga89] Agarwal, A., Horowitz, M., and Hennessy, J., ‘‘An Analytical Cache Model’’. *ACM Trans. on Comp. Sys.*, Vol.7, no.2, May 1989, pp. 184-215.

[Arvi87] Arvind, and Ianucci, R.A., ‘‘Two Fundamental Issues in Multiprocessing’’. *Proc. of DFVLR - Conf. 1987 on Parallel Proc. in Sc. and Eng.*, West Germany, June 1987, pp. 61-88.

[Arvi88] Arvind, Culler, D.E., and Maa, G.K., ‘‘Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs’’. *The International Journal of Supercomputer Applications*, Vol.2, No.3, November 1988.

[Clar83] Clark, D.W., ‘‘Cache Performance in the VAX-11/780’’. *ACM Trans. Comp. Sys.*, Vol.1, No.1, February 1983, pp. 24-37.

[Egge88] Eggers, S.J., and Katz, R.H., ‘‘A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation’’. *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1988, pp. 373-383.

[Good83] Goodman, J.R., ‘‘Using Cache Memory to Reduce processor-Memory Traffic’’. *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, 1983.

[Hals88] Halstead, R.H., Jr., and Fujita, T., ‘‘MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing’’. *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1988, pp. 443-451.

[Ianu88] Ianucci, R.A., ‘‘Toward a Dataflow / von Neumann Hybrid Architecture’’. *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1988, pp. 131-140.

[Nikh88] Nikhil, R.S. and Arvind, ‘‘Can Dataflow Subsume von Neumann Computing?’’. *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, Jerusalem, Israel, June 1989.

[Russ78] Russell, R.M., ‘‘The CRAY-1 Computer System’’. *Comm. of the ACM*, Vol.21, No.1, January 1978. pp. 63-72.

[Saav90] Saavedra-Barrera, R.H., and Culler, D., ‘‘An Analytical Solution for a Markov Chain Modeling Multithread Execution’’, University of California, Berkeley, technical report in preparation.

[Smit82] Smith, A.J., ‘‘Cache Memories’’. *ACM Computing Surveys*, Vol.14, No.3, September 1982, pp. 473-530.

[Smit78] Smith, B.J., ‘‘A Pipelined, Shared Resource MIMD Computer’’. *1978 Int. Conf. on Parallel Proc.*, 1978, pp. 6-8.

[Thie89] Thiebaut, D., ‘‘On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio’’. *IEEE Trans. on Computers*, Vol.38, No.7, July 1989, pp. 1012-1026.

[This88] Thistle, M.R., and Smith, B.J., ‘‘A Processor Architecture for Horizon’’. *Supercomputing ’88*, Florida, October 1988, pp. 35-40.

[Webe89] Weber, W., and Gupta A., ‘‘Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results’’. *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, Jerusalem, Israel, June 1989, pp. 273-280.