RC 20238 (10/23/95)
Computer Science

IBM Research Report

LIMITS ON THE PERFORMANCE BENEFITS OF MULTITHREADING AND PREFETCHING

Beng-Hong Lim

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Ricardo Bianchini

Federal University of Rio de Janeiro
Rio de Janeiro, RJ 21945-970
Brazil

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Limits on the Performance Benefits of Multithreading and Prefetching

Beng-Hong Lim
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
bhlim@watson.ibm.com

Ricardo Bianchini
Federal University of Rio de Janeiro
Rio de Janeiro, RJ 21945-970 Brazil
ricardo@cos.ufrj.br

October 23, 1995

Abstract

The performance benefits of multithreading and software prefetching depend heavily on hardware and application-specific parameters, such as the multiprocessor's remote memory access latency and the cache miss rate of parallel applications. This paper considers the relationship between these parameters and the maximum performance improvement achievable through multithreading and software prefetching. Specifically, the paper presents new analytical models and experimental measurements of several parallel applications on the MIT Alewife multiprocessor and uses them to predict the benefits of these latency-tolerating techniques. For the first time, both techniques are evaluated on a real machine as opposed to simulations. The models determine the regions in the parameter space where the techniques may be most effective, while the measurements determine the regions where applications actually lie. We find that these regions do not always overlap significantly.

The multithreading model explains why 2–4 contexts is sufficient to achieve most of this technique's potential benefit in the current generation of shared-memory multiprocessors. For these multiprocessors, the maximum improvement in running time from multithreading is less than 10% for most of the applications in this paper. The model also shows that multithreading can significantly improve the performance of the same applications in future multiprocessors with much longer latencies, provided sufficient hardware contexts exist to tolerate the longer latencies. Reducing context-switch overhead will not be crucial in these multiprocessors.

The software prefetching model shows that this technique can improve performance over a wide range of parameters. On current multiprocessors, allowing up to 4 prefetches to be outstanding is sufficient to achieve most of the potential benefit of this technique. Prefetching can improve the running time of several of the applications by as much as 20–50% even on current multiprocessors. The rest of the applications may also benefit in future multiprocessors. A comparison of the two models shows that software prefetching has a significant advantage over multithreading for machines with low remote access latencies and/or applications with high remote cache miss rates because a prefetch instruction typically consumes less time than a context-switch.

1 Introduction

The high latency of remote memory accesses is a major impediment to good application performance on scalable cache-coherent multiprocessors. Several techniques have been proposed for coping with remote-memory latency in these machines, including *block multithreading* and *software prefetching*. These techniques overlap communication latencies with useful computation to reduce processor stalls. Block multithreading, or multithreading for short, achieves this overlap by switching between processor-resident threads on cache misses that involve remote communication (remote cache misses). Software prefetching, or prefetching for short, achieves this overlap by requesting cache blocks that will likely lead to cache misses in advance.

The performance benefits of multithreading and prefetching depend on parameters such as the multiprocessor's remote cache miss latency, an application's remote cache miss rate, and the amount of useful computation performed by a processor in between consecutive remote cache misses. We define the last parameter as the application's *run-length*.

This paper develops simple, but useful analytical models that use these parameters to predict the maximum performance improvement achievable through multithreading and prefetching. It introduces the notion of *speedup* to characterize the performance benefit of a latency-tolerating technique, where speedup is defined as the ratio of program execution times without and with the technique. The multithreading model extends Agarwal's model of processor utilization [2] to predict the maximum speedup due to multithreading. The prefetching model is new, and predicts both utilization and speedup due to prefetching.

The models are useful for investigating the different design tradeoffs associated with multithreading and prefetching, to understand the effect that technological changes have on performance, and to predict the potential benefits of each latency-tolerating technique. They determine the range of architectural and application parameters, such as run-length, remote cache miss latency and context-switch overhead, where multithreading and prefetching are most likely to improve performance.

As a concrete example of the utility of the models, this paper applies them towards evaluating multithreading and prefetching on the MIT Alewife Machine [1], a distributed shared-memory multiprocessor that implements both techniques. For the first time, both multithreading and prefetching are evaluated on an actual working machine as opposed to simulations. We measure the run-lengths of a number of parallel applications (including SPLASH [27] and NAS parallel benchmarks [5]) and use the models to predict the potential benefits of multithreading and prefetching for these applications. To corroborate the models' predictions, we measure the execution times of versions of the applications that use the latency-tolerating techniques.

The multithreading model explains why 2–4 contexts is sufficient to achieve most of this technique's potential benefit in the current generation of shared-memory multiprocessors. For these multiprocessors, the maximum improvement in running time from multithreading is less than 10% for most of the applications in this paper. The model also shows that multithreading can significantly improve the performance of the same applications in future multiprocessors with much longer latencies, provided sufficient hardware contexts exist to tolerate the longer latencies. Reducing context-switch overhead will not be crucial in these multiprocessors.

The prefetching model shows that this technique can improve performance over a wide range of parameters. On current multiprocessors, allowing up to 4 prefetches to be outstanding is sufficient to achieve most of the potential benefit of this technique. Prefetching can improve the running time of several of the applications by as much as 20–50%, even on current multiprocessors. The rest of the applications may also benefit in future multiprocessors.

A comparison of the two models shows that software prefetching has a significant advantage over multithreading for machines with low remote access latencies and/or applications with high remote cache miss rates because a prefetch instruction typically consumes less time than a context-switch. However, prefetching does not always realize its higher potential of improving performance over multithreading because it requires prefetch instructions to be inserted at the right places. Furthermore, long run-lengths and latencies reduce the negative impact of context-switch overhead in multithreading.

This paper makes the following contributions.

- It presents models of multithreading and prefetching that can be used to determine the usefulness of latency-tolerating techniques in the hardware design and application space.
- The multithreading model exposes the limits of multithreading by considering speedup as well as processor utilization, unlike previous models that focus only on processor utilization.
- As far as we know, this paper is the first to present a processor utilization and speedup model for software prefetching.
- It presents an analytical and experimental comparison of multithreading and prefetching on a single multiprocessor platform. This allows us to isolate the differences between the two techniques while keeping other parameters constant.
- It introduces the concept of a "sweet spot", a criterion for determining whether a particular latency-tolerating technique is useful for a particular set of hardware parameters and application characteristics.
- It shows, through a comparison of sweet spots against measured run-lengths of a representative set of applications on a real distributed shared-memory multiprocessor, that some of the applications we examine cannot benefit significantly from either technique in the current-generation of shared-memory multiprocessors.

The remainder of this paper is organized as follows. The next section presents some background on multithreading and prefetching. Section 3 presents our models of the two techniques. Section 4, describes the MIT Alewife Machine and our application workload, and presents the results of our analyses and experiments on Alewife. Section 5 considers the potential benefits of multithreading and prefetching for other architectures. Finally, Section 6 summarizes the conclusions drawn from this research.

2 Background

Multithreading and prefetching have been the focus of work on tolerating communication latency for many years. Multithreaded processors have been proposed or used in several machines, such as the HEP [28], Monsoon [25], the Tera MTA [4], McGill MTA [16], and MIT Alewife [1]. [7] and [17] describe the history and status of multithreaded architectures and research.

Most of the earlier multithreaded architectures follow an aggressive fine-grain approach to multithreading, where a large number of contexts are available and a context-switch takes a small number of cycles. The main criticism of this approach is poor single-thread performance. To address this limitation, Alewife introduced the idea of block multithreading, where caches reduce the number of remote cache misses and context-switches occur only on such misses. This allows Alewife to use a less aggressive implementation of multithreading [3].

Some recent hardware designs [18, 21] try to provide both single-cycle context-switches and high single-thread performance by allowing instructions from multiple threads to coexist and interleave arbitrarily in the processor pipeline. The Tera MTA [4] takes an extreme approach by providing each processor with 128 hardware contexts that support up to 128 active threads. At each clock cycle, the processor executes an instruction from one of the active threads, effectively performing a zero-cycle context switch. A remote memory access takes about 70 cycles on a 256-processor configuration. Although Tera does not have caches, its aggressive design may allow it to achieve close to full processor utilization, *provided* enough threads are available, and enough network bandwidth exists.

Previous experimental research in evaluating multithreading performance has shown that multithreading can be effective at tolerating memory and synchronization latencies for some applications [15, 30, 29]. Previous analytical research [2, 26, 24, 13] has focused on modeling processor utilization and predicting the number of contexts necessary to achieve a reasonably high level of processor utilization. In contrast, this paper combines analytical models and experimental measurements and applies the analysis in a novel way. It considers the performance improvement due to multithreading and defines a range of run-lengths for which multithreading is profitable. It compares this range to the run-lengths observed for real applications.

Prefetching has also been considered in many flavors, with hardware [10, 12], software [23, 8], and hybrid [6] approaches. Software prefetching, where a compiler inserts prefetch instructions for blocks that will likely miss in the cache when referenced, is widely accepted as one of the most efficient prefetching techniques. This technique requires little hardware support and usually incurs the overhead of a single instruction per cache block prefetched. Several multiprocessors provide software prefetching, among them Stanford DASH [22], the KSR1 [14], the Cray T3D [11] and MIT Alewife.

As far as we know, performance studies of software prefetching have all been experimental [23, 8, 6]. These studies indicate sizable performance improvements and consistently better performance than other prefetching strategies. We significantly extend these contributions by modeling software prefetching both in terms of processor utilization and speedup, defining a range of run-lengths for which software prefetching achieves its best results, and comparing this range to the set of experimentally observed run-lengths.

Empirical comparisons of block multithreading and software prefetching are extremely uncommon. Gupta *et al.* [15] find that prefetching leads to better performance when the techniques are applied either in isolation or in combination with relaxed consistency. They show that with an aggressive 4-cycle context-switch overhead, combining relaxed consistency and multithreading achieves good performance. Our study uses both modeling and measurement

p	Number of contexts on a processor
C	Context-switch overhead (time to switch between processor-resident threads)
$T(p)$	Remote cache miss latency (time to service a cache miss involving remote nodes)
$t(p)$	Average run-length (time between remote cache misses)
$U(p)$	Processor utilization (percentage of time doing useful computation)
$S(p)$	Speedup due to multithreading ($U(p)/U(1)$)

Table 1: *Definitions of the parameters used in the model for multithreading.*

to provide insight into previous results. The models allow parameters to be varied easily and their effects observed, thus enabling more thorough comparisons.

3 Performance Models of Multithreading and Prefetching

In this section, we develop analytical models of multithreading and prefetching. The models predict processor utilization and speedup, based on machine and application parameters such as remote cache miss latencies and application run-lengths. They allow us to evaluate and compare the performance of both latency-tolerating techniques.

The analysis makes the following assumptions:

- Multithreaded processors switch contexts only on remote cache misses, and software prefetches are issued only for blocks that would otherwise miss in the caches.
- Remote cache miss latencies and run-lengths are fixed throughout the execution interval of an application that we wish to model.
- The models assume that both multithreading and prefetching result in longer remote cache miss latencies and shorter run-lengths due to increased memory traffic and cache pollution. However, they do not model the magnitude of the differences in latencies and run-lengths.
- The analysis ignores the impact of multithreading and prefetching on program synchronization performance and on scheduling.

3.1 A Performance Model for Multithreading

Previous research has presented sophisticated models of multithreading performance [2, 26, 24, 13] that focus on predicting processor utilization as accurately as possible. These models attempt to account for the effect of multithreading on other system parameters such as cache and network behavior.

In contrast, this research uses a simple model of processor utilization and extends it to include the speedup of an application due to multithreading. The resulting model predicts the limits on the performance gain achievable through multithreading, and allows us to characterize more precisely the regime in which multithreading may significantly benefit an application. While more sophisticated utilization models may yield tighter upper bounds on the benefit of

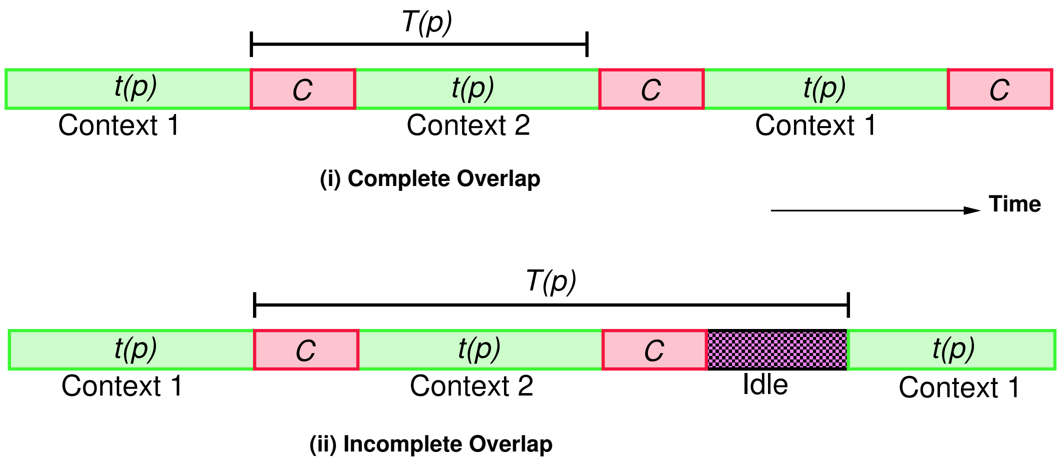

Figure 1: *Time lines illustrating the overlap of communication latency $T(p)$, with computation $t(p)$, when multithreading with $p = 2$ contexts. In the second case, $T(p)$ is too long to be completely hidden and results in some idle processor cycles.*

multithreading, we find that for our purposes, the added accuracy is not worth the additional complexity.

The main parameters that affect the performance of multithreaded processors are the number of contexts p, the context-switch overhead C, the latency of remote cache misses $T(p)$, and the average run-length $t(p)$. Our model uses these parameters to predict processor utilization, $U(p)$, and the speedup due to multithreading, $S(p)$. Table 1 summarizes these parameters.

As Figure 1 illustrates, there are two cases to consider when modeling the processor utilization of a multithreaded processor. In the first case, enough contexts exist to overlap remote cache miss latency completely. This occurs when $T(p) < (p-1)t(p) + pC$. In the second case, not enough contexts exist so that the processor suffers some idle time. Thus, the utilization of a multithreaded processor is given by

$$U(p) = \begin{cases} \frac{t(p)}{t(p)+C} & \text{if } T(p) < (p-1)t(p) + pC \\ \frac{pt(p)}{t(p)+T(p)} & \text{otherwise} \end{cases} \tag{1}$$

This is essentially the same model for processor utilization as presented in [2]. Although the model is useful for determining processor utilization, we carry the analysis one step further and derive an upper bound on the speedup that can be achieved due to multithreading, $S(p)$. To do this, we compute the ratio $U(p)/U(1)$.

The processor utilization for a non-multithreaded processor is

$$U(1) = \frac{t(1)}{t(1) + T(1)}$$

As observed in [2], multithreading may shorten run-lengths and increase remote memory latencies. That is, $t(p) \leq t(1)$ due to cache interference and finer computation grain sizes, and $T(p) \geq T(1)$ due to higher network and memory contention. With these inequalities, the maximum performance gain achievable from using a multithreaded processor with p contexts is

$$S(p) = U(p)/U(1) \quad \leq \quad \begin{cases} \frac{t(p)+T(p)}{t(p)+C} & \text{if } T(p) < (p-1)t(p) + pC \\ \\ p & \text{otherwise} \end{cases} \tag{2}$$

Equations 1 and 2 provide a useful tool for evaluating the potential benefits of multithreading for a given machine architecture. The machine's hardware architecture determines the values of $T(p)$ and C. With these, we can determine the range of run-lengths, $t(p)$, such that multithreading will result in *both* a significant speedup *and* a reasonable level of processor utilization in order to justify the hardware support for multithreading. We define that range of run-lengths to be the *sweet spot* of multithreading for that architecture.

In Sections 4 and 5, we will use the multithreading model to evaluate multithreading in Alewife and other architectures. We now turn to developing a corresponding model for processor utilization and speedup due to prefetching.

3.2 A Performance Model for Prefetching

This section develops a model for prefetching that is analogous to the multithreading model just presented. Unlike multithreading, software prefetching relies on a user or compiler to insert explicit prefetch instructions for data cache blocks that would miss otherwise. Often, such a task is program-dependent. Thus, we need to identify the type of programs we are interested in modeling.

In the following analysis, we focus on regular loops as a natural candidate for prefetching. We find that a number of the applications we consider in this paper contain loops that can be optimized through prefetching. Furthermore, such loops comprise a significant fraction of the total execution time of those applications.

The prototypical loop that we optimize with prefetching looks like:

```
for (i = 0; i < N; i++)
   compute(i);
```

A prefetching compiler would transform such a loop to a piece of code similar to:

```
prefetch(0, p);                            /* prefetch p blocks for first iteration */
for (i = 0; i < N−step; i = i+step) {      /* note increased step size */
   prefetch(i+step, p);                    /* prefetch p blocks for iter i+step to i+2*step−1 */
   for (j = 0; j < step; j++)              /* compute iters i to i+step−1 */
      compute(i*step + j);
}
for (j = 0; j < step; j ++)                /* compute last 'step' iterations */
   compute(i*step + j);
```

p	Number of cache blocks prefetched at a time
c	Prefetch overhead (time to execute a prefetch instruction)
$T(p)$	Remote cache miss latency (time to service a cache miss involving remote nodes)
t	Average computation per prefetched cache block
$U(p)$	Processor utilization with p prefetches at a time
$S(p)$	Speedup due to prefetching $(U(p)/U(0))$

Table 2: *Definitions of the parameters used in the model for prefetching.*

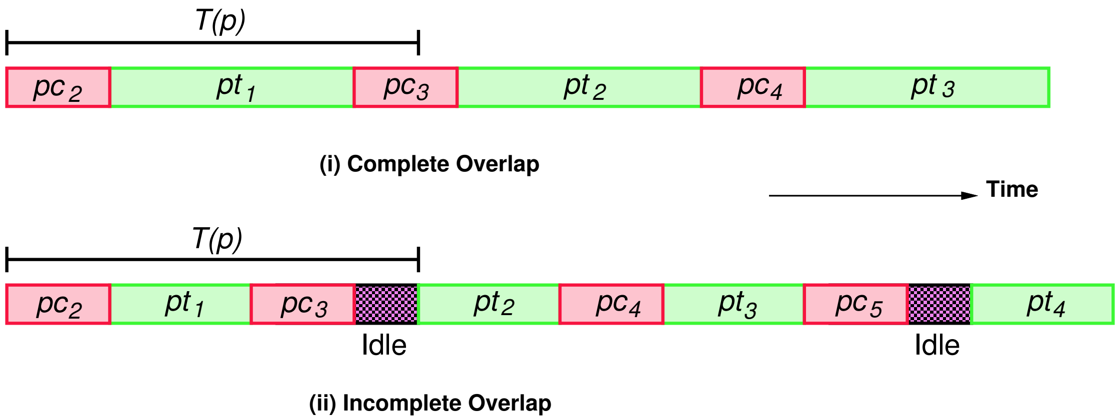

Figure 2: *Time lines illustrating the overlap of communication latency $T(p)$, with computation pt, when prefetching p blocks at a time. pc_i prefetches cache blocks for computation block pt_i. In the second case, $T(p)$ is too long to be completely hidden and results in some idle processor cycles in every other computation block.*

The parameters involved in modeling the behavior of the above prefetching loop are the average number of cache blocks prefetched at a time, p, the overhead of each prefetch instruction, c, the latency of remote cache misses, $T(p)$, and the average amount of computation per prefetched block, t. This last parameter is equivalent to the run-length $t(p)$ in the multithreading model. Table 2 summarizes these parameters.

Prefetching allows for greater flexibility when trying to overlap communication and computation, since the compiler or user can schedule the prefetches according to the remote access latency of the machine and the amount of work done per iteration of each loop in the program. This intelligent scheduling of prefetches can be implemented by adjusting p and the iteration step size appropriately (see code segments above).

However, the optimal number of blocks to prefetch at a time also depends on the cache space available to the processor; blocks brought into the cache by early prefetches may be replaced before being used. Another limitation is the number of prefetched blocks that can fit in a prefetch/transaction buffer. Under these constraints, the amount of useful work that can be overlapped with communication may be insufficient to hide memory latency completely.

Figure 2 illustrates the behavior of the prefetching loop. The execution alternates between prefetch and computation intervals, with each prefetch accessing cache blocks that will be used one computation block ahead[1]. As with the multithreading model, there are two cases to consider. In the first case, enough blocks are prefetched to provide enough computation to overlap remote memory access latency completely. In the second case, the computation block is too short to avoid processor idle time. Thus, the resulting processor utilization is given by

$$U(p) \quad = \quad \begin{cases} \frac{t}{t+c} & \text{if } T(p) < pt + 2pc \\ \\ \frac{2pt}{pt+T(p)} & \text{otherwise} \end{cases} \tag{3}$$

To derive an upper bound on the speedup due to prefetching, we compute the ratio $U(p)/U(0)$. The processor utilization without prefetching is

$$U(0) = \frac{t}{t + T(0)}$$

Again, we assume $T(p) \geq T(0)$, so that the maximum performance gain achievable from prefetching is

$$S(p) = U(p)/U(0) \quad \leq \quad \begin{cases} \frac{t+T(p)}{t+c} & \text{if } T(p) < pt + 2pc \\ \\ \frac{2p(t+T(p))}{pt+T(p)} & \text{otherwise} \end{cases} \tag{4}$$

In a manner similar to the multithreading model, Equations 3 and 4 provide a useful tool for evaluating the potential benefits of prefetching. For a given machine architecture, we can determine the range of t such that prefetching results in *both* a significant speedup *and* a reasonable level of processor utilization in order to justify the hardware support for prefetching.

The following section applies these models towards an evaluation of multithreading and prefetching in the MIT Alewife Machine.

4 Evaluation of Multithreading and Prefetching on Alewife

The MIT Alewife Machine is a prototype implementation of the Alewife architecture [1]. Most importantly, for the purposes of this paper, it provides support for both multithreading and prefetching. Thus, it presents an ideal platform for applying our models towards evaluating both latency-tolerating techniques.

The analysis proceeds as follows. First, we use the latency, context-switch, and prefetch overhead parameters of Alewife to determine the corresponding sweet spots for each of the techniques. Next, we measure the run-lengths of a number of parallel applications executing on Alewife to find out whether they lie within the sweet spots. Last, to corroborate the models, we run a subset of the applications with multithreading and prefetching enabled, and measure the resulting speedups.

We begin with an overview of the Alewife architecture.

[1]While it is possible to have loops prefetch more than one computation block ahead, we can always transform such loops into an equivalent loop that prefetches a single computation block ahead.

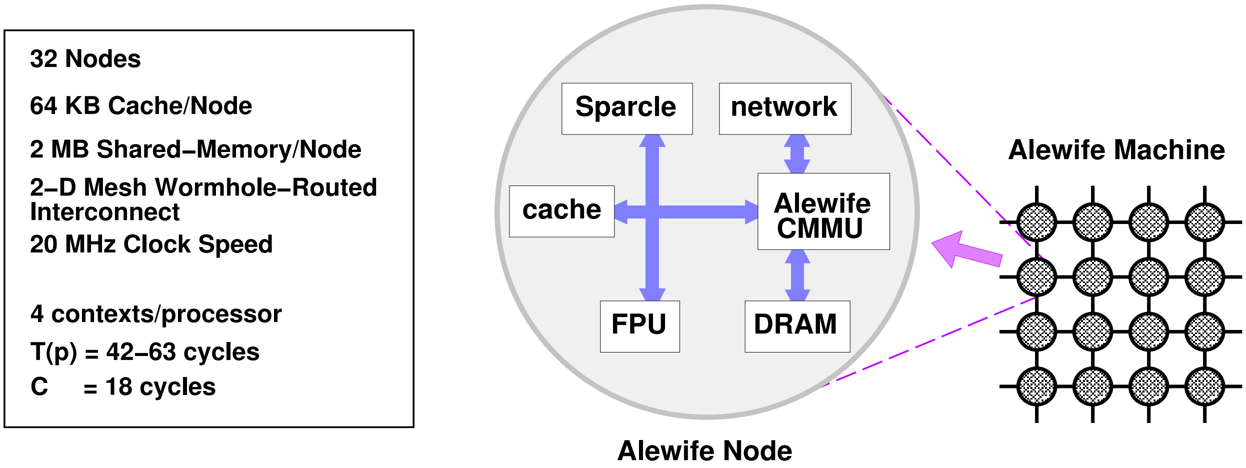

Figure 3: *Architecture of the MIT Alewife Machine.*

4.1 The MIT Alewife Machine

Figure 3 presents an overview of the architecture and organization of Alewife. Each node consists of a Sparcle processor [3] and an associated floating point unit with up to 4 processor contexts, 64K bytes of direct-mapped cache, 8M bytes of DRAM, an Elko-series mesh routing chip (EMRC) from Caltech, and a custom-designed Communication and Memory Management Unit (CMMU). The EMRC network routers use wormhole routing and are connected in a 2-D mesh topology. The single-chip CMMU implements Alewife's scalable LimitLESS cache coherence protocol [9], and provides Sparcle with a low-latency interface to the network. To support both multithreading and prefetching, the caches are lockup free [19].

Alewife supports multithreading via Sparcle's register windows and fast trap handling. A cache miss involving remote communication generates a trap that performs an 18-cycle context-switch to another register window[2]. Alewife supports both read and write prefetching by using special Sparcle memory loads and stores, and a transaction buffer in the CMMU to store prefetch data [20]. The transaction buffer also doubles as a victim cache and is necessary for supporting split-phase memory transactions.

Alewife's most important parameters for our models are its 18-cycle context-switch overhead, 2-cycle prefetch instruction overhead, and average remote cache miss latency of 53 cycles. (A two-party cache miss takes 42 cycles and a three-party cache miss takes 63 cycles.) Thus, we assume that $C = 18$, $c = 2$, and $T(p) = 53$.

4.2 Model Predictions for Multithreading and Prefetching in Alewife

Figures 4 and 5 present the results of the multithreading and prefetching models with Alewife's parameters. Each graph plots simultaneously both processor utilization and speedup as a function of run-length.

These graphs show that for both techniques, speedups are significant for short run-lengths.

[2]Sparcle actually supports a 14-cycle context-switch, but a minor bug in the current version of the CMMU requires a 4-cycle work-around.

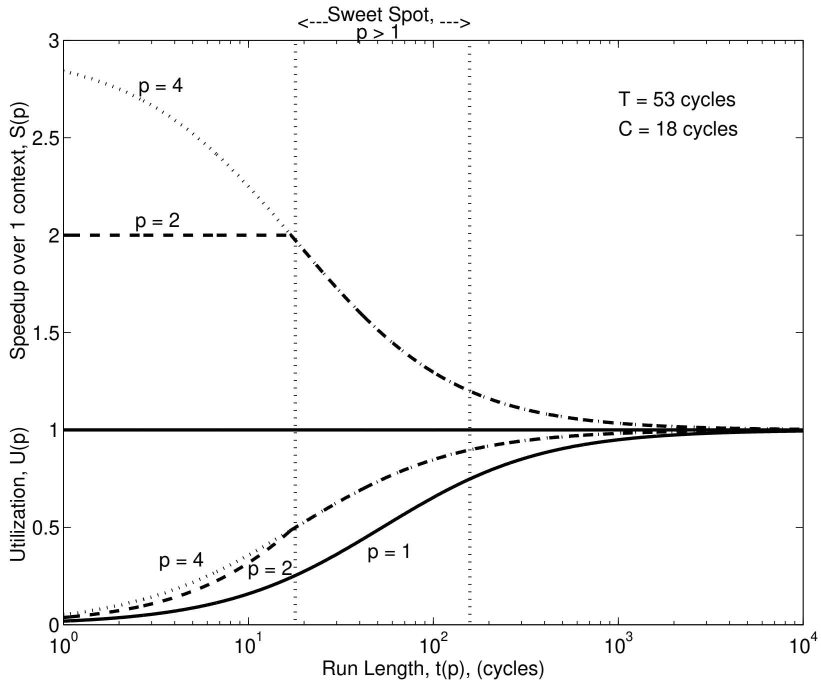

Figure 4: *Modeling the performance of multithreading on Alewife.*

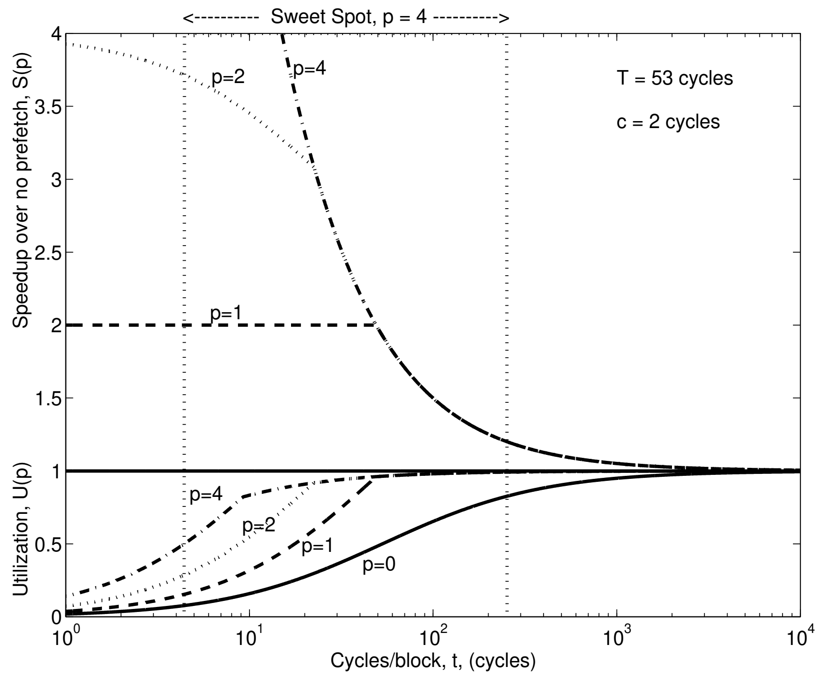

Figure 5: *Modeling the performance of prefetching on Alewife.*

However, processor utilization is extremely low for these same run-lengths. As expected, long run-lengths lead to high levels of processor utilization. However, speedups due to the latency tolerating techniques are very limited. These observations highlight the fact that the sweet spot for both techniques lies somewhere in between the extremes of run-lengths. Determining the exact bounds of the sweet spot depends on one's criteria of minimally acceptable utilization and speedup levels.

In our evaluation, we define the sweet spot to be between the shortest run-length that yields at least 50% processor utilization (left bound) and the longest run-length that yields at least 20% speedup potential (right bound). Although we chose this range of values based on our own criteria of acceptable values for utilization and speedup, others are free to decide on different criteria. The essential point here is that there is a bounded range of run-lengths where a latency-tolerating technique is profitable.

In contrast, previous research has focused on whether multithreading or prefetching can achieve high levels of processor utilization without considering whether they are operating in a regime where the additional speedup from latency tolerance is significant enough to care about.

Figure 4 shows that Alewife has a very narrow multithreading sweet spot of between about 18 and 105 cycles. This narrow range can be attributed to Alewife's rather short remote cache miss latencies. Another interesting observation is that within the sweet spot, two contexts are sufficient to attain all of the potential gains of multithreading. In a sense, the Alewife hardware designers have aggressively minimized memory latencies to a point where multithreading has rather limited benefits.

Figure 5 shows that Alewife's prefetching sweet spot ranges from 4 to 250 cycles. It also shows that the number of cache blocks that can be prefetched at a time, p, affects the performance of prefetching significantly. The reason for this is that the amount of computation that can be used to hide communication latency is expected to grow linearly with p. Within the sweet spot, allowing up to 4 prefetches to be outstanding is sufficient to achieve most of the potential gains of prefetching.

The figures show that the prefetching sweet spot is wider than the multithreading sweet spot mainly because of the lower overhead of prefetching: 2 cycles versus 18 cycles. For short run-lengths of less than 100 cycles, prefetching speedups are significantly better than multithreading speedups. For longer run-lengths however, the overhead of a context-switch becomes less significant and the two techniques perform comparably.

The analysis thus far poses an unanswered question: where do typical applications lie along the spectrum of run-lengths? The next section presents measurements of application run-lengths on the Alewife machine to determine if they fall within the sweet spot.

4.3 Application Performance on Alewife

In order to determine the range of run-lengths found in real applications on a real machine, we use a 32-processor Alewife multiprocessor to study a large number of parallel applications, including the SPLASH and NAS parallel benchmarks[3] and a number of engineering kernels.

[3]Refer to the SPLASH and NAS documents for descriptions of the benchmarks.

Program	Input Size
Barnes-Hut	16K bodies, 4 iterations
Cholesky	3,948 $\times$ 3,948 floats, 56,934 non-zeros (TK15)
LocusRoute	3,817 wires, 20 routing channels (Primary2)
MP3D	18K particles, 6 iterations
Water	125 molecules, 10 iterations
Multigrid	56 $\times$ 56 $\times$ 56 floats
CG	1,400 $\times$ 1,400 doubles, 78,148 non-zeros
EM3D	20,000 nodes, 20% remote neighbors
FFT	80,000 floats
Gauss	512 $\times$ 512 floats
SOR	512 $\times$ 512 floats, 50 iterations
SOR-Dyn	512 $\times$ 512 floats, 50 iterations
BLU	512 $\times$ 512 floats, 16 x 16 blocks
MergeSort	64,000 integers

Table 3: Main application input parameters.

Table 3 lists the applications and their corresponding input parameters. Among the engineering kernels, EM3D simulates electromagnetic wave propagation through 3-D objects, FFT performs a Fast Fourier Transform, Gauss performs Gaussian elimination, BLU performs blocked LU decomposition, MergeSort sorts a list of integers, and SOR performs red-black Jacobi relaxation. SOR-Dyn is a dynamically partitioned and scheduled version of SOR.

To measure multithreading run-lengths, $t(p)$, we take advantage of Alewife's built-in statistics hardware that includes event and cycle counters. We set up the statistics hardware to count the number of remote cache misses and the execution time of each application. Assuming an average remote cache miss time of 53 cycles, we derive a value for the average run-length of an application.

To measure prefetching run-lengths, t, we hand-instrument each loop that has been optimized with prefetching. The instrumented loops measure the actual execution time for each prefetched cache block. Subtracting two cycles for each prefetch instruction from the execution time yields the value of t.

Figures 6 and 7 present multithreading and prefetching run-lengths, respectively, in applications running on 32 Alewife processors[4]. Note that the prefetching run-lengths are quite different from the multithreading run-lengths, although in the models they are comparable parameters. This difference is an artifact of the way we measured the run-lengths, as described in the previous two paragraphs: multithreading run-lengths are measured over the entire application, while prefetching run-lengths are measured only at loops optimized with prefetching. It is important to note that this strategy reflects the common practical uses of these techniques; multithreading is implemented in hardware and is active throughout execution, while prefetching is implemented by the compiler or programmer for optimizing loops.

We find that most of the applications have long multithreading run-lengths of 200 to 2000 cycles that lie outside of the sweet spot. This does not bode well for the prospect of speeding

[4]Results from experiments on 16-processor configurations do not differ significantly from 32 processors.

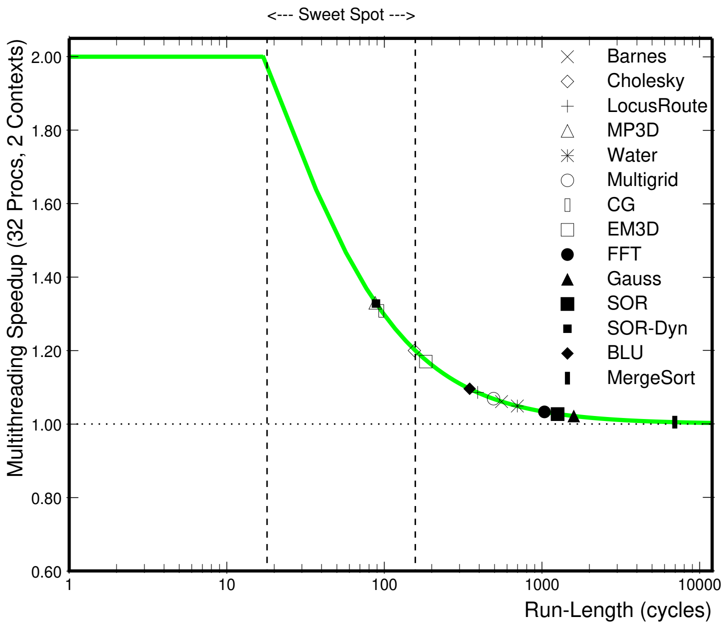

Figure 6: *Potential speedup based on multithreading run-lengths on Alewife.*

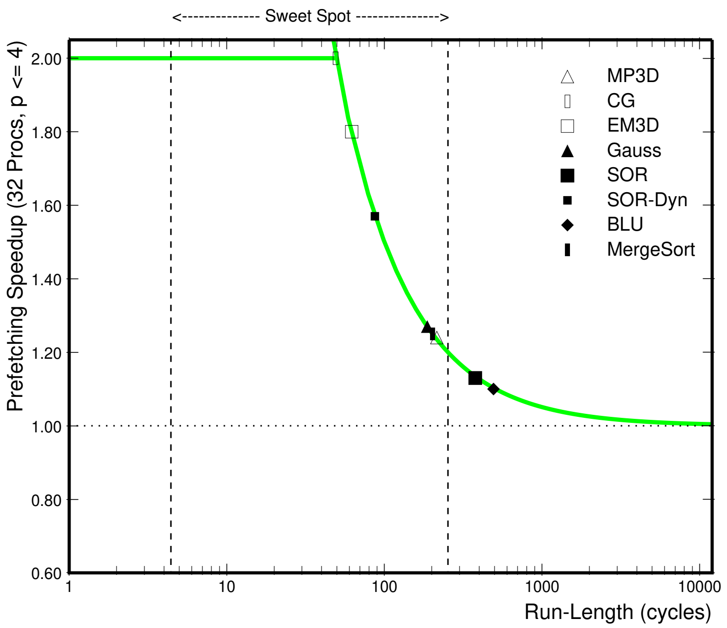

Figure 7: *Potential speedup based on prefetching run-lengths on Alewife.*

up these applications with multithreading. Only MP3D, CG, and Cholesky lie within the sweet spot and may benefit from multithreading. The run-lengths for prefetching are shorter, and range from 50 to 500 cycles, with MP3D, CG, EM3D, Gauss, SOR-Dyn and MergeSort all falling within the sweet spot.

The most important observation here is that for Alewife's parameters, most of the applications operate on a suboptimal part of the performance space for multithreading. This implies that multithreading is not likely to yield much benefit for the applications we considered. The main reason for this is Alewife's short remote cache miss latencies relative to the application run-lengths. The prospect is better for prefetching, but only for applications where we can identify loops for optimization.

In order to validate our models' predictions of the upper bound of application speedups, we execute a subset of the applications on Alewife using the latency-tolerating techniques and measure the change in execution times. For multithreading, we statically schedule a thread on each hardware context, using two contexts on each processor[5]. This prevents scheduling overhead from diminishing any potential speedup. We produced multithreading versions of only a subset of our applications, due to minor incompatibilities between Alewife's thread-based model of parallel execution and the benchmarks' process-fork-based model. For prefetching, we identify loops in a subset of the applications that are amenable to optimization with prefetch instructions, and modified them in accordance with the description in Section 3.

Figures 8 and 9 present the speedup results. We see that the applications respect the models' upper bounds on the achievable speedup for each technique. Some applications come close to achieving optimal performance, but many others do not as a result of factors such as cache interference and/or contention. We also find that the latency-tolerating techniques sometimes affect the synchronization and scheduling behavior of the application adversely. Characterizing or avoiding performance degradation from such factors is an interesting research area but is beyond the scope of this paper.

The model and experimental results for Alewife show that the benefits of multithreading and prefetching are rather limited for the current generation of shared-memory machine architectures like Alewife and Stanford DASH, where remote cache miss latencies are relatively short. The model can be easily applied to other machine architectures, and the next section investigates the effect of different architectural parameters such as latency and context-switch/prefetch overhead on the performance of each technique.

5 Potential Benefits on Other Architectures

The latency of remote cache misses, $T(p)$, the overhead of context switching, C, and the overhead of prefetching, c, are significant factors on the performance of multithreading and prefetching. In order to investigate the potential of these techniques for current and future architectures, this section models the sensitivity of utilization and speedup on each of these parameters.

[5] We also experimented with three contexts, but found no significant performance difference.

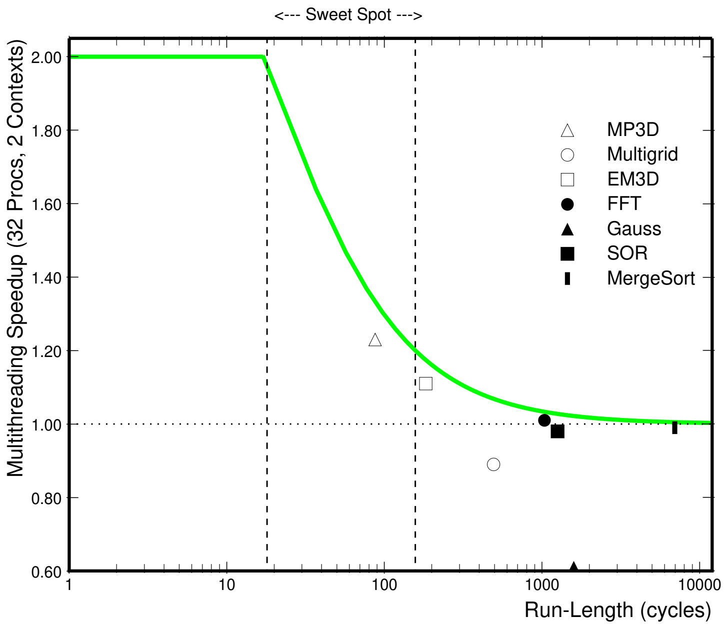

Figure 8: *Measured application speedup on Alewife due to multithreading.*

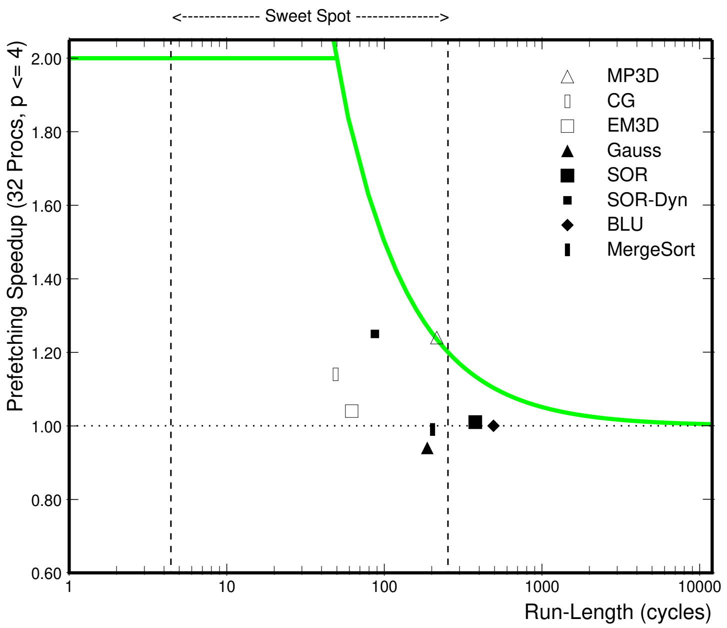

Figure 9: *Measured application speedup on Alewife due to prefetching.*

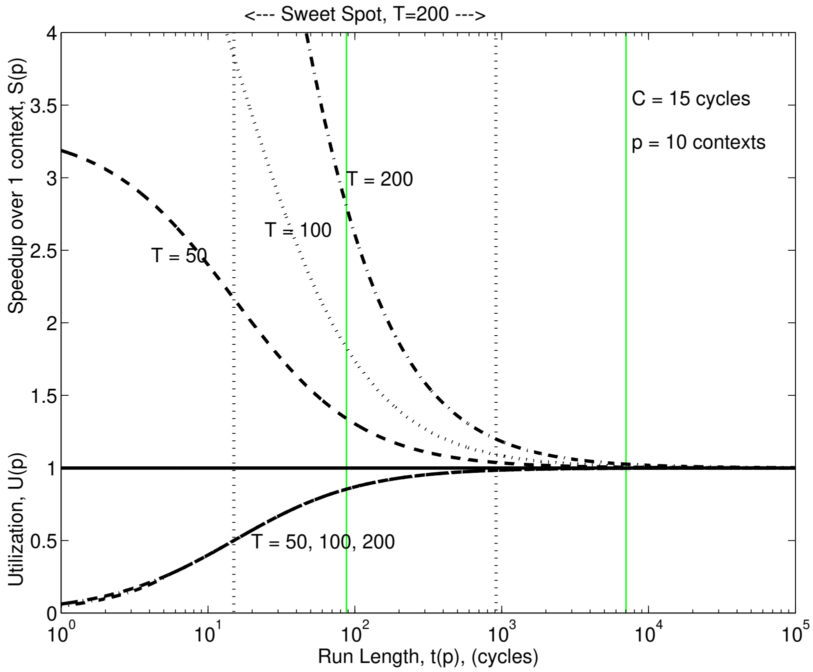

Figure 10: *Effect of remote cache miss latency on multithreading. Vertical lines delimit the sweet spot and the range of application run-lengths.*

5.1 Effect of Remote Cache Miss Latency

The latency of remote cache misses determines the amount of communication per request that must be tolerated by multithreading and prefetching. Previous research show that longer latencies require a larger number of contexts or prefetch buffers to tolerate them. In this work, the models show that longer latencies also increase the potential for improving performance via latency-tolerating techniques.

Figures 10 and 11 model the effect of varying remote cache miss latencies, assuming that enough contexts or prefetch buffers are available to tolerate the latencies. The vertical lines in each graph delimit the sweet spot and the range of application run-lengths. We see that longer latencies tend to widen the sweet spot and increase the overlap with the application run-lengths.

Figure 10 shows the effect of varying remote cache miss latencies on multithreading, with $p = 10$ and $C = 15$. The graph shows that latency has a significant impact on the performance of multithreading: even though the processor utilization is virtually the same for all the latencies we considered, the potential speedup due to multithreading is much greater with longer latencies, and the sweet spot extends to longer run-lengths.

Figure 11 shows the effect of varying remote cache miss latencies on prefetching, with $p = 10$ and $c = 2$. The graph shows that increasing latency has the effect of shifting the entire sweet spot towards longer run-lengths. For the run-lengths in the applications in this study, the effect of latency on utilization is negligible, while the effect on speedup is much more pronounced.

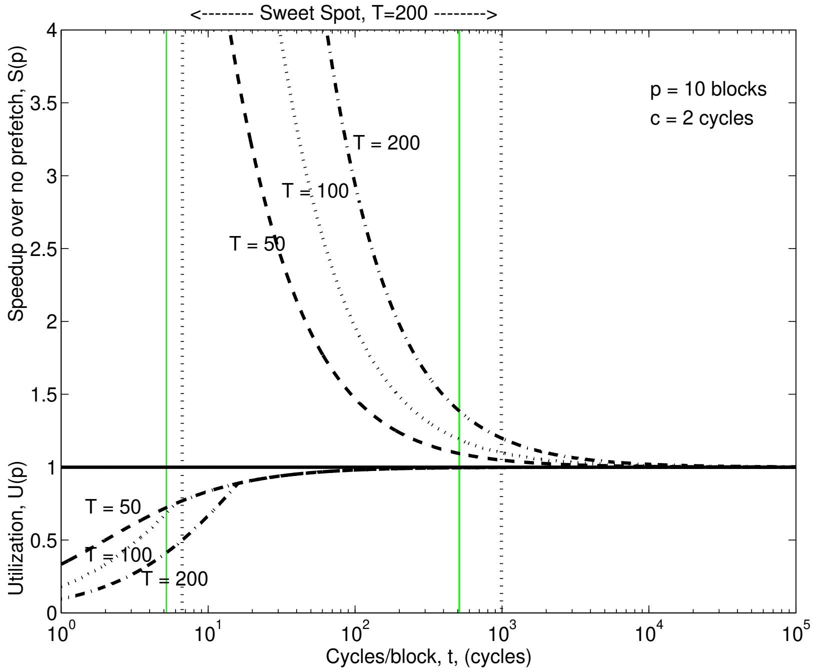

Figure 11: *Effect of remote cache miss latency on prefetching. Vertical lines delimit the sweet spot and the range of application run-lengths.*

These results imply that a large number of the SPLASH and NAS benchmarks would benefit from multithreading and/or prefetching on machines with remote latencies longer than 200 cycles. Recall that only three of the applications have run-lengths longer than 1000 cycles. The rest of the applications would fall within the sweet spots.

5.2 Effect of the Overhead of Multithreading and Prefetching

Another parameter that affects performance is the overhead associated with each technique: the context-switch overhead for multithreading and the prefetch instruction overhead for prefetching. Assuming enough contexts or prefetch buffers to cover the latency, lower overheads tend to extend the sweet spot towards shorter run-lengths. However, there is little or no impact on the sweet spot at longer run-lengths. Figures 12 and 13 illustrate this effect.

Figure 12 shows the effect of varying context-switch overhead on multithreading, with $p = 10$ and $T = 50$. The graph shows that context-switch overhead significantly impacts performance for short run-lengths. A lower overhead tends to extend the sweet spot towards shorter run-lengths. However, context-switch overhead does not have a significant impact within the range of application run-lengths we found.

With higher latencies, the models predict that context-switch overhead has an even smaller impact on performance. This result implies that Alewife's context-switch overhead is small enough for the applications we considered, and a more aggressive implementation is necessary only for applications with run-lengths shorter than about 100 cycles.

Figure 13 shows the effect of varying prefetch instruction overhead on prefetching, with

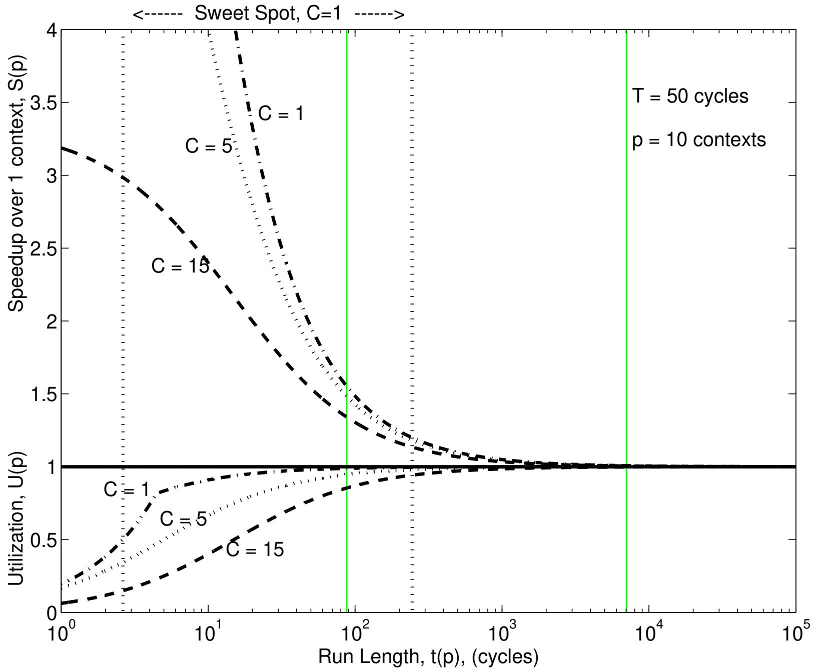

Figure 12: *Effect of context-switch overhead on multithreading. Vertical lines delimit the sweet spot and the range of application run-lengths.*

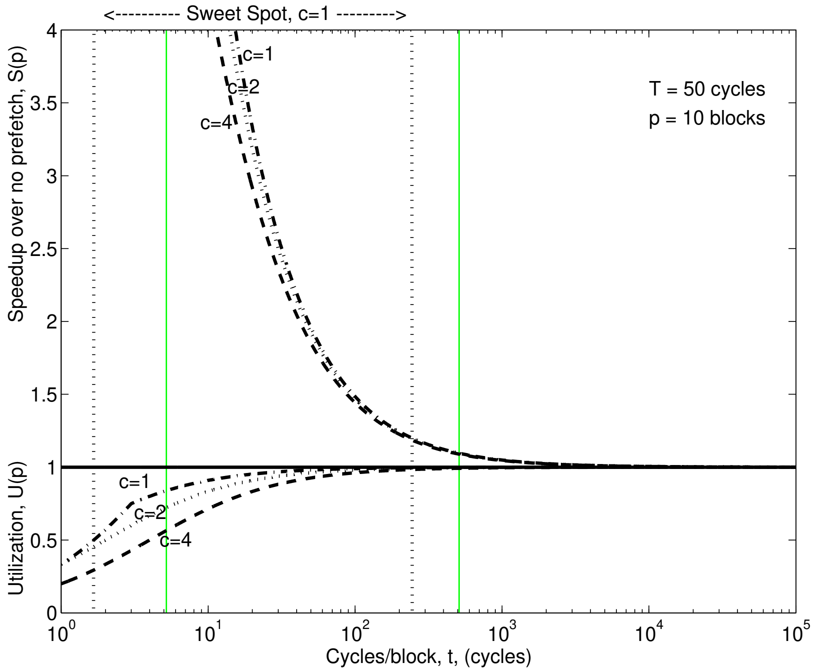

Figure 13: *Effect of prefetch instruction overhead on prefetching. Vertical lines delimit the sweet spot and the range of application run-lengths.*

$p = 10$ and $T = 50$. The graph shows that prefetch overhead significantly impacts performance for short run-lengths. We also see that prefetch overhead has very little effect for run-lengths that are longer than 50 cycles. As with multithreading, this implies that prefetch overhead is not a significant factor for the applications we considered.

The figure also shows the interesting result that even a 2-cycle run-length is within the prefetching sweet spot for this set of parameters. This result means that even an application with no locality whatsoever can achieve reasonable performance with prefetching[6], provided that enough prefetch buffers exist.

To summarize, these results show that lower overheads extend the sweet spot towards shorter run-lengths, in contrast with longer latencies that extend the sweet spot towards longer run-lengths. Since the application run-lengths in our experiments either lie outside the sweet spots or only overlap with their rightmost region, overhead makes little difference on performance for the applications in our suite.

5.3 Implications for Other Architectures

We conclude this section by speculating on the implications of our models and analysis on the architecture of other computing platforms.

A recently promoted alternative to custom-designed multiprocessors for parallel computing is to use commodity workstations connected via high-speed networks. Multithreading and prefetching may have a larger role to play here. Such a platform is likely to have remote cache miss latencies in the order of 10 μs. Assuming a 5 ns cycle time, this translates to a latency of about 2000 cycles. At such latencies, our models predict that multithreading and prefetching are likely to improve performance significantly. Since latencies are so high, these techniques could even be implemented entirely in software, handling memory pages as opposed to cache blocks. The hardware would only need to support non-blocking memory loads and stores.

Multiprocessor latency-hiding techniques have also been suggested as methods to improve performance in uniprocessor systems. As advances in processor technology outstrip advances in memory technology and local cache-miss latencies become more significant. It is conceivable that local cache misses will take upwards of 50 cycles to service in the next generation of high-speed processors, close to today's multiprocessors' remote cache miss latencies. However, uniprocessor local cache misses and are likely to be much more frequent shorter than multiprocessor remote cache misses. Thus, on such a platform, context-switch and prefetch overheads will be significant and will need to be reduced as much as possible.

6 Summary and Conclusions

This paper introduces analytical models of processor utilization and speedup for two latency-tolerating techniques: multithreading and prefetching. The models, based on hardware and application characteristics such as memory latencies and application run-lengths, show the limits as well as the benefits of these techniques. Using the models, we define sweet spots; the

[6] A similar effect can be observed in Figure 12, but hardware support for a 1-cycle context-switch is more expensive than for a 1-cycle prefetch instruction overhead.

ideal operating conditions under which the techniques may improve performance significantly. We compare the sweet spots against the ranges of run-lengths from a representative set of real applications running on the MIT Alewife machine. This comparison shows that very few applications can benefit significantly from multithreading in current multiprocessors, while some but not all applications can benefit from prefetching in these multiprocessors.

An evaluation of the effect of future technological changes shows that remote cache miss latencies have a much greater impact on the performance of the techniques than their associated overheads. A comparison of the two models shows that prefetching has an advantage over multithreading because a context-switch usually consumes more processor cycles than a prefetch instruction. However, prefetching requires user intervention or compiler technology to identify prefetch points.

Our models provide a methodology for architects and programmers to evaluate if multithreading and prefetching will be effective for a particular set of architectural parameters and target application domain. With the models, we find that prefetching is preferable over multithreading for machines with low remote cache miss latencies and/or applications with poor locality and consequently short run-lengths. The performance of both techniques is comparable for machines with high remote cache miss latencies and/or applications with good locality. For the applications in our suite, prefetching is the technique of choice for current multiprocessors, while either multithreading or prefetching can be successful for future multiprocessors.

7 Acknowledgments

We would like to thank the Alewife group for allowing us to run our experiments on the Alewife machine, and especially John Kubiatowicz for help with the Alewife statistics hardware. The Alewife project is funded in part by ARPA contract # N00014-94-1-09885, in part by NSF Experimental Systems grant # MIP-9012773, and in part by a NSF Presidential Young Investigator Award.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, June 1995.

[2] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, Amsterdam, June 1990. ACM.

[5] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[6] Ricardo Bianchini and Thomas J. LeBlanc. A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors. Technical Report TR 515, Department of Computer Science, University of Rochester, May 1994.

[7] Gregory T. Byrd and Mark A. Holliday. Multithreaded Processor Architectures. *IEEE Spectrum*, pages 38–46, August 1995.

[8] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, Boston, MA, April 1991.

[9] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[10] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Prefetching Schemes. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, Chicago, IL, April 1994. ACM.

[11] Cray T3D System Architecture Overview, 1993. Cray Research, Inc.

[12] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[13] Pradeep K. Dubey, Arvind Krishna, and Mark S. Squillante. Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures. Computer Science RC 19661, IBM, July 1994.

[14] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of the 38th Annual IEEE Computer Society Computer Conference (COMPCON)*, pages 284–294, San Francisco, CA, 1993. IEEE.

[15] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 254–263, Toronto, Canada, May 1991. ACM.

[16] Herbert Hum *et al.* The Multi-Threaded Architecture Multiprocessor. Technical Report ACAPS Technical Memo 88, McGill University School of Computer Science, December 1994.

[17] Robert A. Iannucci, editor. *Multithreaded Computer Architecture - A Summary of the State of the Art.* Kluwer Academic Publishers, 1994.

[18] Stephen W. Keckler and William J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 202–213, Gold Coast, Australia, June 1992. IEEE.

[19] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 81–87, June 1981.

[20] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, Boston, MA, October 1992. ACM.

[21] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 308–318, San Jose, CA, October 1994. ACM.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[23] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[24] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Agarwal. Analysis of Multithreaded Architectures with Distributed Shared Memory. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, Dallas, TX, 1993. IEEE.

[25] G. Papadopoulos and D. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 82–91, June 1990.

[26] R.H. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–177, July 1990.

[27] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[28] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photooptical Instrumentation Engineers*, 298:241–248, 1981.

[29] Radhika Thekkath and Susan Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 328–337, San Jose, CA, October 1994. ACM.

[30] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 120–130, San Diego, CA, May 1993. IEEE.