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
COPPE Systems Engineering/UFRJ
Rio de Janeiro, RJ 21945-970 Brazil
ricardo@cos.ufrj.br

Abstract

This paper presents new analytical models of the performance benefits of multithreading and prefetching, and experimental measurements of parallel applications on the MIT Alewife multiprocessor. For the first time, both techniques are evaluated on a real machine as opposed to simulations. The models determine the region in the parameter space where the techniques are most effective, while the measurements determine the region where the applications lie. We find that these regions do not always overlap significantly.

The multithreading model shows that only 2–4 contexts are necessary to maximize this technique's potential benefit in current multiprocessors. Multithreading improves execution time by less than 10% for most of the applications that we examined. The model also shows that multithreading can significantly improve the performance of the same applications in multiprocessors with longer latencies. Reducing context-switch overhead is not crucial.

The software prefetching model shows that allowing 4 outstanding prefetches is sufficient to achieve most of this technique's potential benefit on current multiprocessors. Prefetching improves performance over a wide range of parameters, and improves execution time by as much as 20–50% even on current multiprocessors. The two models show that prefetching has a significant advantage over multithreading for machines with low memory latencies and/or applications with high cache miss rates because a prefetch instruction 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 this problem, including *block multithreading* and *software prefetching*. These techniques overlap communication with useful computation to reduce processor stalls. Block multithreading, or multithreading for short, switches between processor-resident threads on cache misses that involve remote communication. Software prefetching, or prefetching for short, requests in advance cache blocks that would otherwise lead to cache misses.

SIGMETRICS '96 5/96 PA, USA
© 1996 ACM 0-89791-793-6/96/0005...$3.50

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

This paper develops simple but useful analytical models that predict the maximum performance benefit achievable through multithreading and prefetching. It introduces the notion of *gain* to characterize the benefit, where gain is defined as the ratio of program execution times without and with the technique. The multithreading model extends Agarwal's processor utilization model [1] to predict the gain due to multithreading. The prefetching model is new, and predicts both utilization and gain due to prefetching.

The models are useful for investigating the design tradeoffs associated with multithreading and prefetching, to understand the effect of technological changes, 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.

This paper applies the models towards evaluating multithreading and prefetching on the MIT Alewife Machine [2], a distributed shared-memory multiprocessor that implements both techniques. For the first time, both techniques are evaluated on an actual machine as opposed to simulations. We measure the run-lengths of a number of applications and use the models to predict the potential benefits of multithreading and prefetching for these applications. To corroborate the models' predictions, we measure the actual performance of the applications with the latency-tolerating techniques.

The multithreading model shows that 2–4 contexts are sufficient to maximize this technique's potential benefit in the current generation of shared-memory multiprocessors. Multithreading improves execution time by 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 outstanding prefetches is sufficient to achieve most of the potential benefit of this technique. Prefetching can improve the execution 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 that determine the usefulness of multithreading and prefetching in the hardware design and application space. The models expose performance limits by considering gain as well as processor utilization. As far as we know, this paper is the first to present a processor utilization and gain 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 if a latency-tolerating technique is useful for a particular set of hardware parameters and application characteristics. A comparison of sweet spots against run-lengths of a representative set of applications on Alewife shows that some of the applications cannot benefit significantly from either technique in current multiprocessors.

The remainder of this paper is organized as follows. The next section presents background on multithreading and prefetching. Section 3 presents our models of the two techniques. Section 4 describes Alewife and our application workload, and presents the results of our analyses and experiments. Section 5 considers the benefits on 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 [25], Monsoon [22], the Tera MTA [4], McGill MTA [14], and MIT Alewife [2]. A description of the history and status of multithreaded architectures and research appears in [7] and [15].

Early multithreaded architectures take an aggressive fine-grain approach that switches contexts at each instruction. This requires a large number of contexts and very low overhead context switches. The main drawback 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 a less aggressive implementation of multithreading [3].

Some recent architectures [16, 18] promise both single-cycle context-switches and high single-thread performance by allowing instructions from multiple threads to interleave arbitrarily in the processor pipeline. The Tera MTA [4] takes an extreme approach by providing 128 contexts on each processor. At each cycle, the processor executes an instruction from one of the contexts. Tera does not have caches, and requires a large number of concurrent threads and high network bandwidth to achieve high processor utilization.

Previous experimental research on multithreading performance shows that multithreading is effective at tolerating memory latencies for some applications [13, 26]. Previous analytical research [1, 23, 21, 11] focuses on modeling processor utilization and predicting the number of contexts needed for good processor utilization. In contrast, this paper combines analytical models and experimental measurements in a novel way. It defines a range of run-lengths where multithreading is profitable, and compares this range to the run-lengths in real applications.

Prefetching has been considered in many flavors, with hardware [9, 10], software [20, 8], and hybrid [6] approaches. Software prefetching, where a compiler inserts prefetch instructions for blocks ahead of actual use, is widely accepted as one of the most efficient prefetching techniques. It requires little hardware support and usually incurs the overhead of a single instruction per cache block prefetched. Several machines provide software prefetching, including Stanford DASH [19], the KSR1 [12], and MIT Alewife.

As far as we know, performance studies of software prefetching have all been experimental [20, 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 gain, defining a range of run-lengths where prefetching is profitable, and comparing this range to experimentally observed run-lengths.

Empirical comparisons of multithreading and prefetching are extremely uncommon. Gupta *et al.* [13] find that prefetching leads to better performance when applied either in isolation or in combination with relaxed consistency. They show that multithreading with an aggressive 4-cycle context-switch overhead, combined with relaxed consistency achieves good performance. Our study uses both modeling and measurement to provide insight into such previous results by allowing parameters to be varied easily and their effects observed.

3 Performance Models of Multithreading and Prefetching

This section develops analytical models of multithreading and prefetching that predict upper bounds on processor utilization and gain, based on machine and application parameters such as remote cache miss latencies and application run-lengths. The models make the following assumptions:

- Multithreaded processors switch contexts only on remote cache misses. Software prefetches are issued only for blocks that would otherwise cause a remote cache miss.
- Remote cache miss latencies are constant within the section of an application that we wish to model.
- Multithreading and prefetching result in longer remote cache miss latencies and shorter run-lengths due to increased memory traffic and cache pollution.
- The models ignore the impact of multithreading and prefetching on program synchronization performance and scheduling.

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)
$G(p)$	Gain due to multithreading ($U(p)/U(1)$)

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

3.1 A Performance Model for Multithreading

Previous research has presented sophisticated models of multithreading performance 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, we use a simple model of processor utilization and extend it to predict the limits on the gain due to multithreading. This allows us to characterize precisely the regime where multithreading may significantly benefit an application. While more sophisticated utilization models may yield tighter upper bounds on the benefit of multithreading, the added accuracy is not worth the additional complexity for our purposes.

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 gain due to multithreading, $G(p)$. Table 1 summarizes these parameters.

Figure 1 illustrates two cases to consider when modeling the 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

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

This is the same model for processor utilization presented in [1]. We carry the analysis a step further and derive an upper bound on the gain achievable through multithreading, $G(p)$.

The processor utilization for a non-multithreaded processor is

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

As observed in [1], 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 from using a multithreaded processor with p contexts, $G(p) = U(p)/U(1)$, is

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

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

Sections 4 and 5 use the model to evaluate multithreading in Alewife and other architectures. We now develop a corresponding model for processor utilization and gain due to prefetching.

3.2 A Performance Model for Prefetching

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 target for 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 iter 0 */
for (i = 0; i < N-step; i += step) {
  /* prefetch p blocks for iter i+step to i+2*step-1 */
  prefetch(i+step, p);
  /* compute iters i to i+step-1 */
  for (j = 0; j < step; j++) compute(i*step + j);
}
/* compute last 'step' iterations */
for (j = 0; j < step; j ++) compute(i*step + j);
```

The parameters involved in modeling the behavior of the above prefetching loop are the 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 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 per iteration of each loop in

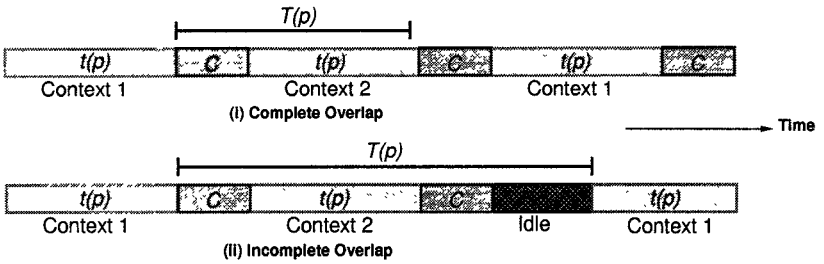

Figure 1: *Time lines illustrating the overlap of communication latency, $T(p)$, with computation, $t(p)$, when multithreading with $p = 2$ contexts. In the second case, $T(p)$ is too long to be completely hidden and results in some idle processor cycles.*

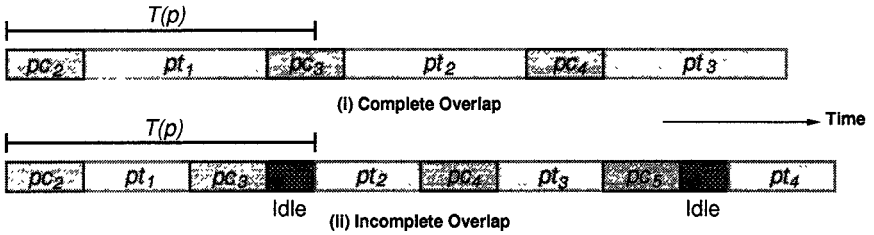

Figure 2: *Time lines illustrating the overlap of communication latency, $T(p)$, with computation, pt, when prefetching p blocks at a time. pc_i prefetches cache blocks for computation block pt_i. In the second case, $T(p)$ is too long to be completely hidden and results in some idle processor cycles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

p	Number of cache blocks prefetched at a time
c	Prefetch overhead (time taken by 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
$G(p)$	Gain due to prefetching ($U(p)/U(0)$)

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

the program. This intelligent scheduling of prefetches adjusts p and the iteration step size appropriately (see code segments above).

The optimal number of blocks to prefetch also depends on the available cache space; blocks prefetched into the cache 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 to be used one computation block ahead.[1] Again, 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. The resulting processor utilization is given by

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

To derive an upper bound on the gain 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 from prefetching, $G(p) = U(p)/U(0)$, is

$$G(p) \leq \begin{cases} \frac{t+T(p)}{t+c} & \text{if } T(p) < pt + 2pc \\ \frac{2p(t+T(p))}{pt+T(p)} & \text{otherwise} \end{cases} \quad (4)$$

In a similar way 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 gain *and* a reasonable level of processor utilization in order to justify hardware support for prefetching.

4 Evaluating Multithreading and Prefetching on Alewife

This section applies our models towards an evaluation of multithreading and prefetching in the MIT Alewife Machine [2]. Alewife provides support for both multithreading and prefetching, and presents an ideal platform for evaluating both techniques. After describing the Alewife architecture, we use Alewife's latency, context-switch, and prefetch overhead parameters to determine the

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

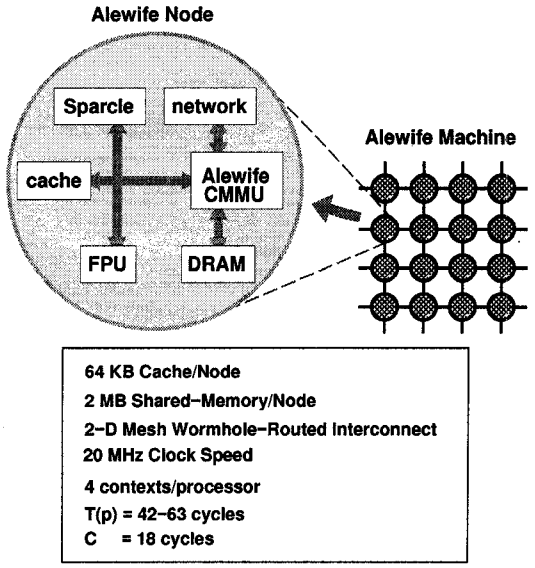

Figure 3: *Architecture of the MIT Alewife Machine.*

sweet spots for each of the techniques. Next, we measure the run-lengths of a number of applications on Alewife to find out if 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 gains.

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 routers use wormhole routing and are connected in a 2-D mesh. The CMMU implements Alewife's scalable cache-coherent shared-memory.

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 [17].

Alewife's 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 Alewife

Figures 4 and 5 present the results of the multithreading and prefetching models with Alewife's parameters. Each graph plots both processor utilization and gain as a function of run-length. These graphs show that for both techniques, gains are significant for short

[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.

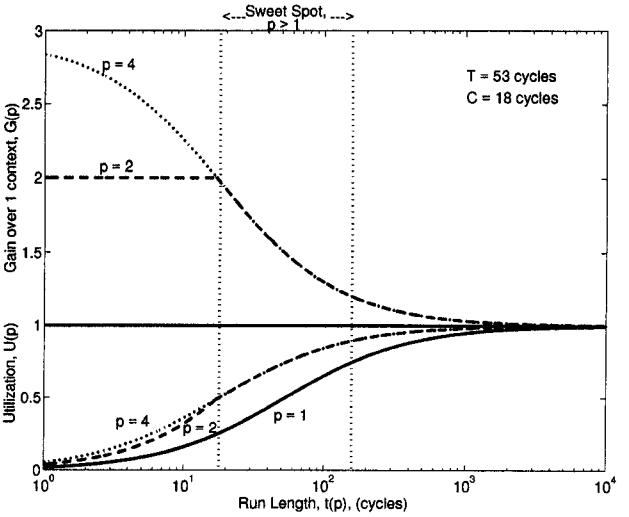

Figure 4: *Modeling the performance of multithreading on Alewife.*

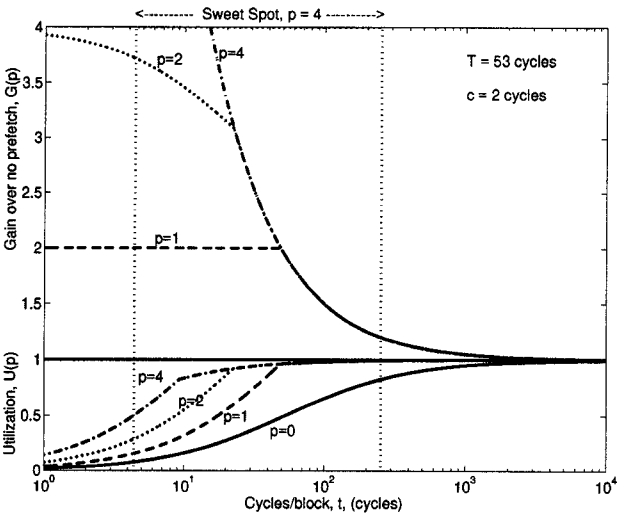

Figure 5: *Modeling the performance of prefetching on Alewife.*

run-lengths. However, processor utilization is extremely low for these same run-lengths. As expected, long run-lengths lead to high levels of processor utilization. However, gains 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.

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% potential gain (right bound). Although we chose this range of values based on our own criteria of acceptable values for utilization and gain, 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 increase processor utilization without considering whether they are operating in a regime where the gain from latency tolerance is significant enough to care about.

Program	Input Size
Barnes-Hut	16K bodies, 4 iterations
Cholesky	3,948 × 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
Appbt	20 × 20 × 20 floats
Multigrid	56 × 56 × 56 floats
CG	1,400 × 1,400 doubles, 78,148 non-zeros
EM3D	20,000 nodes, 20% remote neighbors
FFT	80,000 floats
Gauss	512 × 512 floats
SOR	512 × 512 floats, 50 iterations
SOR-Dyn	512 × 512 floats, 50 iterations
BLU	512 × 512 floats, 16 × 16 blocks
MergeSort	64,000 integers

Table 3: Main application input parameters.

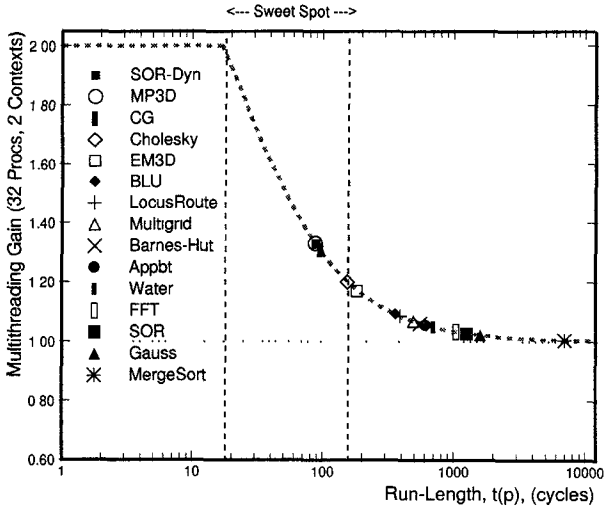

Figure 6: *Potential gain based on multithreading run-lengths.*

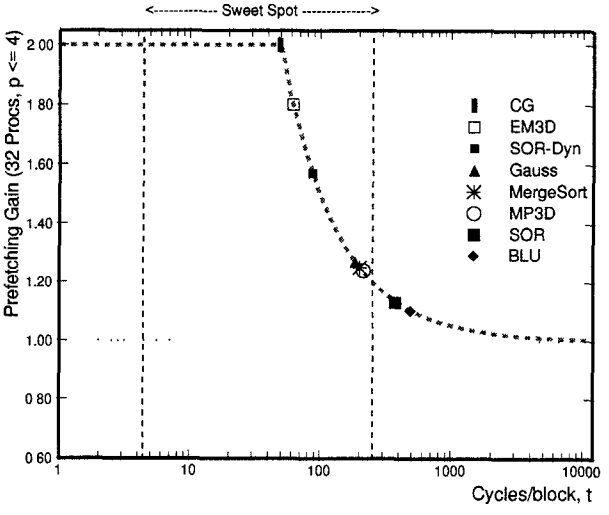

Figure 7: *Potential gain based on prefetching run-lengths.*

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 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 outstanding prefetches is sufficient to achieve most of the potential performance gain of prefetching.

The prefetching sweet spot is wider than the multithreading sweet spot mainly because of the lower overhead of prefetching: 2 versus 18 cycles. For run-lengths of less than 100 cycles, prefetching gains are significantly better than multithreading gains. For longer run-lengths however, the overhead of a context-switch becomes less significant and both 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 [24] and NAS [5] parallel benchmarks and a number of engineering kernels.

Table 3 lists the applications and their input parameters. Please refer to the SPLASH and NAS documents for descriptions of those benchmarks. 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 scheduled version of SOR.

To measure multithreading run-lengths, $t(p)$, we use Alewife's built-in statistics hardware, which includes event and cycle counters, 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 an average value for run-length.

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 the average multithreading and prefetching run-lengths in the applications on 32 Alewife processors,[3] and the corresponding potential gains. The applications are listed in or-

[3] Results on 16-processors do not differ significantly from 32 processors.

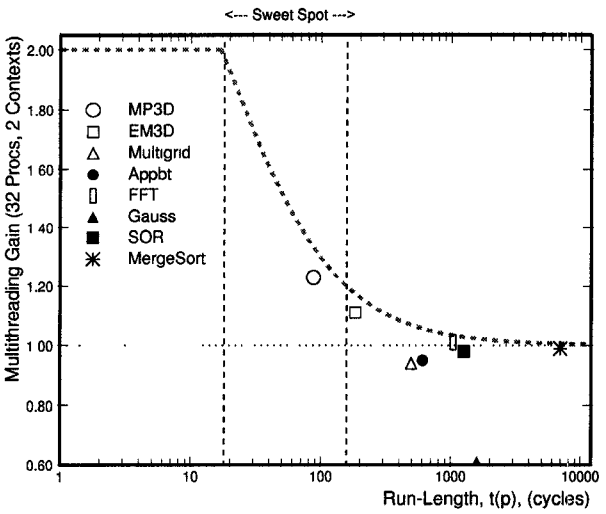

Figure 8: *Measured gain on Alewife due to multithreading.*

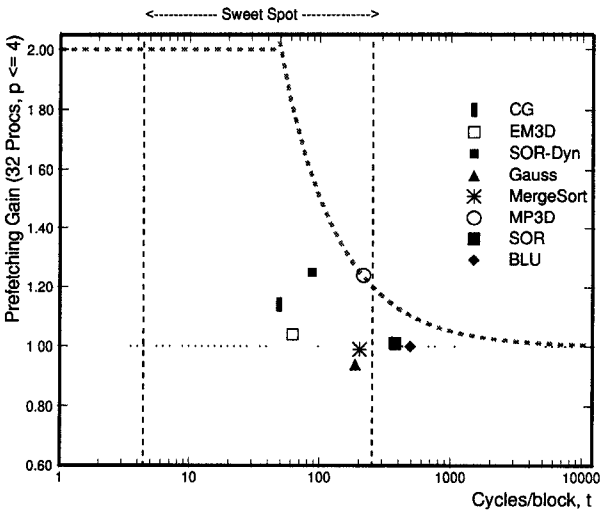

Figure 9: *Measured gain on Alewife due to prefetching.*

der of increasing run-lengths. We should note that using the average run-length in our models yields a correct upper bound on the overall performance of an application even if the application consists of phases with different run lengths. However, we do not provide a proof of this here due to space limitations.

The prefetching run-lengths are generally shorter than the multithreading run-lengths, although they are comparable parameters in the models. One reason for this difference is that multithreading switches contexts only on remote cache misses while prefetches occur for every remote memory access that is *expected* to miss. This causes multithreading run-lengths to tend to be longer than prefetching run-lengths.

Another reason for this difference is that multithreading run-lengths are measured over the entire application, while prefetching run-lengths are measured only at loops optimized with prefetching. This strategy reflects the common practical use of these techniques: multithreading is implemented in hardware and is active throughout execution, while prefetching is implemented by the compiler or programmer to optimize loops. We measured multithreading run-lengths within the loops optimized with prefetching and found that both measurement strategies lead to minor differences in run-lengths, except for MP3D, BLU, and Gauss. In these applications, multithreading run-lengths within the loops are actually longer than the global multithreading run-lengths because cache miss rates outside the loops are higher.

Recall that run-lengths measure the time in between *remote* cache misses and prefetches. Thus, a high local cache miss rate can inflate the run-lengths since the local miss latency in included in the time. Our measurements find that the majority of the applications have local miss rates of well below 5% and that only EM3D has a significant local cache miss rate of 23% on 32 processors.

Most of the applications have long multithreading run-lengths of 200–2000 cycles that lie outside of the sweet spot. This diminishes the prospect of speeding up these applications with multithreading. Only MP3D, CG, and Cholesky lie within the sweet spot and may benefit from multithreading. The prefetching run-lengths are shorter, and range from 50–500 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 parameter 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.

To validate our models' predictions of the upper bound of application gains, we execute a subset of the applications on Alewife with the latency-tolerating techniques and measure the resulting execution times. For multithreading, we statically schedule a thread on each hardware context, with two contexts on each processor.[4] This prevents scheduling overhead from diminishing any potential gain. We produced multithreading versions of only a subset of our applications, due to minor incompatibilities between Alewife's thread-based model and the benchmarks' process-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 results. We see that the applications respect the models' upper bounds on the achievable gain 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 beyond the scope of this paper.

The models and experimental results for Alewife show that multithreading and prefetching yield rather limited benefits for the current generation of shared-memory machine architectures like Alewife and Stanford DASH, where remote cache miss latencies are relatively short. The models 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.

[4] Experiments with three contexts show only minor performance differences.

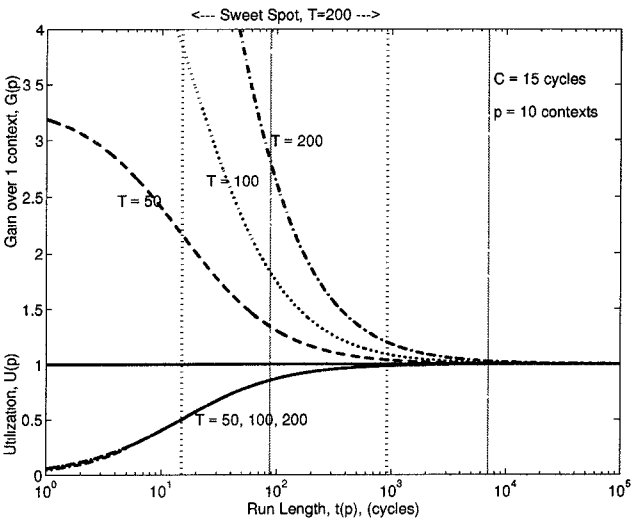

Figure 10: *Effect of remote cache miss latency on multithreading. Vertical lines delimit the sweet spot and the range of application run-lengths.*

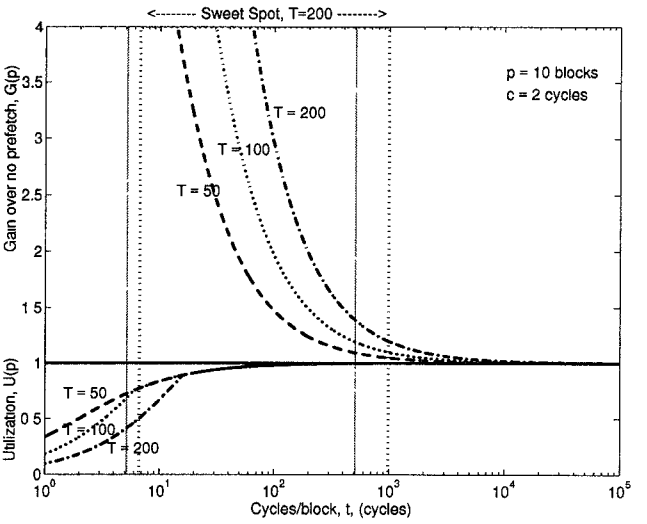

Figure 11: *Effect of remote cache miss latency on prefetching. Vertical lines delimit the sweet spot and the range of application run-lengths.*

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 gain on each of these parameters.

5.1 Effect of Remote Cache Miss Latency

The latency of remote cache misses determines the amount of communication that must be tolerated. Longer latencies require a larger number of contexts or prefetch buffers to tolerate them. 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 enough contexts or prefetch buffers 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 $C = 15$. It 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 gain 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 $c = 2$. It 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 gain is much more pronounced.

These results imply that most of the SPLASH and NAS benchmarks would benefit from multithreading 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 fall within the sweet spots.

5.2 Effect of Multithreading and Prefetching Overhead

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 even less impact on performance. This implies that Alewife's context-switch overhead is small enough for the applications we considered. A more aggressive implementation is necessary only for applications with run-lengths shorter than 100 cycles.

Figure 13 shows the effect of varying prefetch instruction overhead on prefetching, with $p = 10$ 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, 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

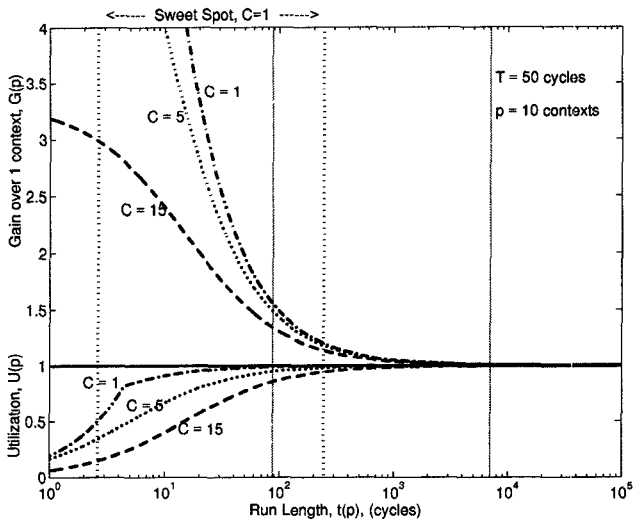

Figure 12: *Effect of context-switch overhead on multithreading. Vertical lines delimit the sweet spot and the range of application run-lengths.*

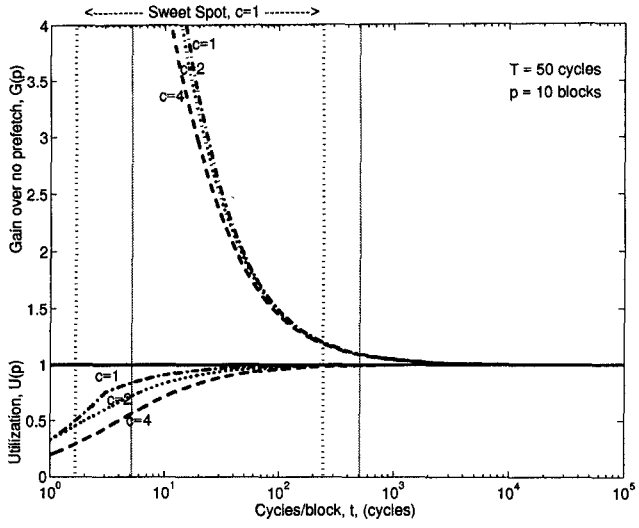

Figure 13: *Effect of prefetch instruction overhead on prefetching. Vertical lines delimit the sweet spot and the range of application run-lengths.*

whatsoever can achieve reasonable performance with prefetching,[5] provided that enough prefetch buffers and network bandwidth exist.

To summarize, 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 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 impact here. Such a platform is likely to have remote memory latencies in the order of 10 μs. Assuming a 5 ns cycle time, this results in 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 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 to improve performance in uniprocessor systems. As advances in processor speeds outstrip advances in memory speeds, local memory latencies become more significant. It is conceivable that cache misses will take upwards of 50 cycles in the next generation of high-speed processors, close to today's multiprocessors' remote memory latencies. Uniprocessor local cache misses are likely to be much more frequent than multiprocessor remote cache misses. On such a platform, context-switch and prefetch overheads will be significant and will need to be reduced as much as possible.

[5] 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.

6 Summary and Conclusions

This paper introduces analytical models of processor utilization and performance gain 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. The models define sweet spots: conditions under which the techniques may improve performance significantly. They provide a method for architects and programmers to evaluate if a particular latency-tolerating technique will be effective for a particular set of architectural parameters and applications.

A comparison of the sweet spots with run-lengths from a representative set of applications running on the MIT Alewife Machine shows that for current multiprocessors, very few of the applications can benefit significantly from multithreading, while some but not all of the applications can benefit from prefetching. The main reason behind this is the relatively short remote cache miss latencies ($<$ 150 cycles) in these machines. With short latencies, prefetching has an advantage over multithreading because a context-switch usually consumes more processor cycles than a prefetch instruction.

An evaluation of the effect of technological changes shows that remote cache miss latencies have a greater impact on the performance benefits of multithreading and prefetching than their associated overheads. A remote cache miss latency of greater than 200 cycles will allow multithreading and prefetching to improve application performance significantly. With longer latencies, the higher overhead of context-switching becomes less of a factor.

A comparison of the two models shows that prefetching is preferable over multithreading for machines with low remote access latencies and/or applications with poor locality and consequently short run-lengths. The performance of both techniques is comparable for applications with high remote access latencies and/or good locality. For the applications in our suite, prefetching is the technique of choice for current multiprocessors, while both multithreading and prefetching can be successful for future multiprocessors.

Acknowledgments

We would like to thank the members of the Alewife group, 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. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] 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*. ACM, June 1995.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. 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] R. Bianchini and T.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] G. Byrd and M. 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*, pages 40–52, Boston, MA, April 1991.

[9] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Prefetching Schemes. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. ACM.

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

[11] P. Dubey, A. Krishna, and M. Squillante. Analytic Performance Modeling for a Spectrum of Multithreaded Processor Architectures. Computer Science RC 19661, IBM, July 1994.

[12] 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.

[13] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, Toronto, Canada, May 1991. ACM.

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

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

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

[17] J. Kubiatowicz, D. Chaiken, and A. 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*, pages 274–284, Boston, MA, October 1992. ACM.

[18] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, San Jose, CA, October 1994. ACM.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] R. Thekkath and S. Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, San Jose, CA, October 1994. ACM.