# Latency Tolerance through Multithreading in Large-Scale Multiprocessors

Kiyoshi Kurihara[*], David Chaiken, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

## Abstract

In large-scale distributed-memory multiprocessors, remote memory accesses suffer significant latencies. Caches help alleviate the memory latency problem by maintaining local copies of frequently used data. However, they cannot eliminate the latency caused by first-time references and invalidations needed to enforce cache coherence. Multithreaded processors tolerate such latencies by rapidly switching between threads when they encounter cache misses. This paper evaluates the effectiveness of multithreading in Alewife, a scalable multiprocessor that is being developed at MIT. For the applications used in this study, multithreading results in a modest 20% improvement in execution time on a 64-processor machine. The impact of multithreading is expected to be far more significant in larger machines, when remote memory latency becomes a dominant term in the performance equation.

[*]Kiyoshi Kurihara is currently at IBM Japan, Ltd. Tokyo, Japan.

## 1 Introduction

Long communication latencies impose limits on the performance attainable by large-scale multiprocessors. As Figure 1 illustrates, processor utilization diminishes when the time a processor spends waiting for responses to remote memory requests is wasted. Idling on remote memory requests wastes not only processor resources but also the bandwidth of the processor interconnection network. When processors remain idle for large periods of time, they do not sustain request rates that fully utilize the available network capacity. Unfortunately, the proportion of time wasted by processors not only increases as machines get larger, but also grows as advances in technology increase the mismatch between the speeds of processors and the speeds of memory and communications components.

Processor utilization can be improved if a processor that would otherwise have to wait for a pending memory or synchronization request can rapidly switch between threads of control, thereby performing useful computation. This strategy attempts to hide the latency of interprocessor communication by allowing multiple outstanding transactions per processor. While previous architectures have implemented multithreading with cycle-by-cycle interleaving of instructions from different processes [11, 12, 16, 21] (termed *fine* multithreading), we use the same name for systems that interleave blocks of instructions from different processes as well [3, 23] (termed *coarse* or *block* multithreading).

Block multithreaded processors do not force a context switch every cycle and can achieve high single-thread performance. They switch between threads only on long-latency memory requests or synchronization attempts. We are using such a block multithreaded processor architecture [3] in the

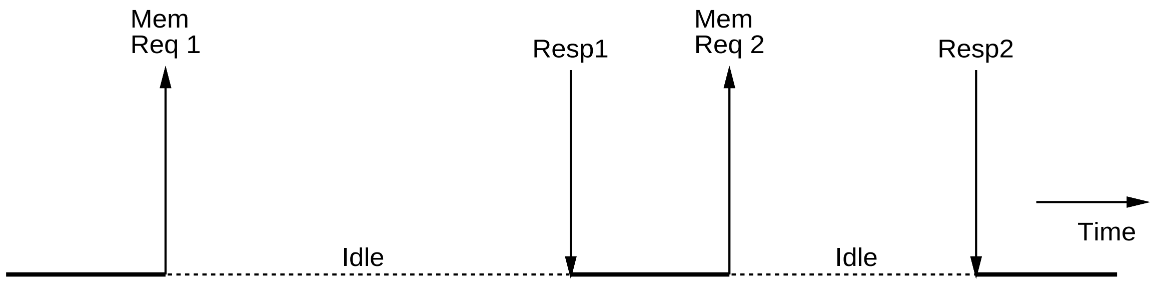

Figure 1: The Long Latency Problem.

Alewife design. Alewife is a large-scale multiprocessor being implemented at MIT; it is described in more detail in the next section. Sparcle, the initial implementation of Alewife's processor, supports multithreading by providing four register sets that hold the state of four threads of control.

Multithreading has several negative side-effects, including increased cache miss rates and higher network contention. In addition, multithreading requires the applications running on a system to provide enough parallelism to sustain several contexts per processor. Performance gains due to hiding communication latency must be traded off against the requirements of a multithreaded system. Our evaluation methodology includes these negative effects while measuring the performance of block-multithreaded processors.

Using multiprocessor simulations, we compare the performance of a system with multithreaded processors to the performance of a system with standard processors. In terms of the speed-up over a single processor system, we find that a 64-processor multithreaded system performs about 20% better than a system with no mechanisms for tolerating communication latency. By presenting an analysis of the cost of various types of memory transactions, we show that the benefits of block multithreading outweigh the negative side-effects of the technique. We conclude that multithreading will provide even higher performance gains in larger systems that suffer from higher interprocessor communication latencies.

The rest of this paper is organized as follows. Section 2 describes the features of Alewife that are relevant to the study of latency tolerance. Section 3 analyzes the problem of communication latency in large-scale multiprocessors and discusses how multithreading can be used to solve the problem. Section 4 presents our simulation methodology, and Section 5 evaluates the simulated performance of Alewife's multithreaded architecture. Section 6 draws conclusions from our simulations and analysis.

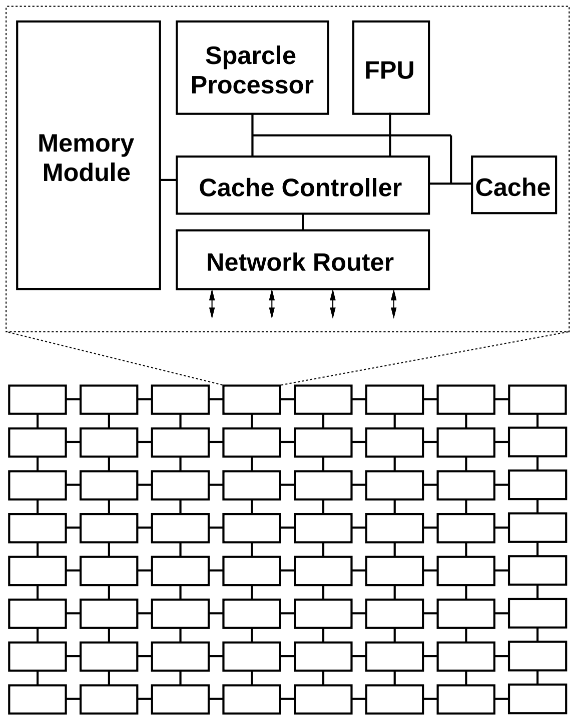

Figure 2: The Structure of the Alewife System.

## 2 The Alewife Architecture

The Alewife group at MIT is designing and implementing a large-scale multiprocessor that uses multiple contexts for latency tolerance. The machine, as depicted in Figure 2, is designed to be physically scalable. The system consists of a set of processing nodes that are connected by a two-dimensional mesh network. This type of network is scalable, because the network bandwidth grows with the number of processors and because the interconnection wire lengths do not depend on the number of nodes in the system. Each node consists of a network router, a Sparcle processor, a floating-point unit, a cache, a cache-memory controller, and a portion of globally-shared memory. The shared memory is distributed to the processing nodes so that the system does not suffer from the bandwidth bottleneck of a single, monolithic memory.

Alewife's cache controllers synthesize a globally shared memory address space. It uses a directory-based cache coherence scheme called the LimitLESS protocol [7], which realizes the performance of full-map directory protocols [5, 22], with the memory overhead of a limited directory protocol [4]. The directory used by the cache coherence protocol is also distributed to the processing nodes.

Latency tolerance in Alewife is part of a layered approach to automatic management of multiprocessor communication locality. Several components in the Alewife system cooperate in automatic minimization of latency. Hardware-managed distributed caches significantly reduce the frequency of remote communications by automatically copying frequently used data locally. The software run-time system allows process and data partitioning, placement, and migration for improving locality. When the system can not avoid a remote memory request and is forced to incur the latency of the interprocessor communication network, the Alewife processors attempt to tolerate this latency through the technique of block multithreading. The next section details the role of multithreading and its implementation.

## 3 Using Multithreading to Tolerate Latency

While caching data reduces the average memory access latency by minimizing the frequency of remote memory accesses, a fraction of memory transactions still require service from remote memory modules. When transactions cause the cache coherence protocol to issue invalidation messages in order to ensure sequential consistency [14], the remote memory access latency is especially high. Transactions that require any invalidations do not complete until all the invalidations are complete. When the resulting remote memory access latency is much longer than the time between memory accesses, processors spend most of their time waiting for memory transactions to be serviced.

This problem, caused by an imbalance between local and remote communication latency, is similar in principle to the problem caused by the imbalance between main memory and i/o device speeds in uniprocessors. The memory versus i/o speed imbalance has been solved by using multi-programming. By time-sharing a processor and multiplexing the communication path between the processor and its i/o devices, their speed differences are efficiently hidden.

Although the absolute communication costs in the uniprocessor system differ from the analogous costs in a multiprocessor, the same type of solution can be applied to solve the problem caused by the imbalance between local and remote memory access speeds in multiprocessors. By allowing each processor to have multiple outstanding memory requests, it is possible for a processor to switch between threads of control in order to mask the latency of remote memory accesses. This solution may be implemented efficiently by designing a processor that can rapidly switch between a number of hardware contexts and allocating one thread of control to each context.

As shown in Figure 3, this technique can increase processor utilization considerably. In the figure, solid lines represent the portion time that a processor executes application code, and dotted lines represent the time that a processor remains idle. The vertical lines designate the requests to remote memory modules, and the corresponding responses. Time flows from left to right. At the beginning of the scenario, Context 1 is active, but becomes idle when it issues Memory Request 1. After a short context switch period (marked with the Sw label), Context 2 becomes active, and runs until it issues Memory Request 2. Again, the processor switches contexts. Since Memory Request 1 was satisfied by Response 1, Context 1 is ready to continue executing. Figure 3 shows one more context switch, which happens when Context 1 issues Memory Request 3. At this point, Context 2 is not yet ready to run, because Memory Request 2 has not yet completed. This illustrates the important point that in order to mask long communication latencies, threads must maximize the time between remote memory accesses.

The type of system depicted in Figure 3 is referred to as a *multithreaded architecture*. The idea of multithreading in multiprocessor systems is not new. The prototypical multithreaded architecture is the HEP [17]. Eight threads reside in each of the HEP's processors, and each thread uses its own dedicated register set. In order to hide long memory access latency and achieve high processor utilization, the system schedules the threads in round-robin fashion. Multithreading is also used in data flow machines [15], and some Lisp-oriented architectures [11].

These multithreaded architectures switch contexts after every instruction execution. Although this switching policy offers the potential of high processor utilization, it results in relatively poor scalar perfor-

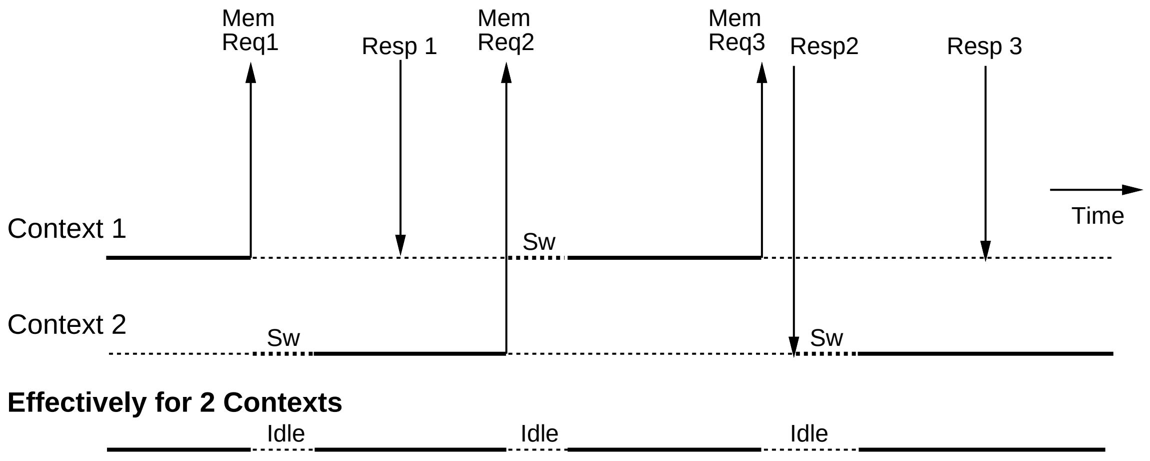

Figure 3: Multithreaded Timeline.

mance observed by any single thread. When there is not enough parallelism to fill all of the hardware contexts, the system performance degrades significantly. For example, in HEP, if there are exactly eight threads, the processor utilization reaches 100%. Otherwise, the processor cycles executed by the idle contexts are wasted. Context switching on a cycle-by-cycle basis reduces system performance on inherently sequential portions of an application.

It is possible to provide reasonable performance on sequential code while still enjoying the merits of multithreading for highly parallel sections of applications. In Alewife's processor architecture, APRIL [3], context switches occur only when a thread executes a memory request that must be serviced by a remote node in the multiprocessor. As long as a context's memory requests hit in the cache or can be serviced by a local memory module, the context continues to execute. This context switching policy allows a single thread to benefit from the maximum performance of the processor.

Sparcle, the initial APRIL implementation, uses the register windows of the SPARC processor [19] to implement multithreading. With a small number of hardware modifications, the register window mechanism can be used to implement both the hardware contexts and the rapid context switch needed for multithreading. Sparcle dedicates one register window to each thread. When a cache miss occurs and a context issues a memory access request that must be serviced remotely, the cache controller traps the processor. The trap routine saves the Program Counter (PC) and Processor Status Register (PSR), switches register sets by setting the Frame Pointer (FP) register, flushes the pipeline, and switches to the next context by setting the Frame Pointer (FP) register to point to a new register window. [3] shows that even with a low-cost implementation, a context switch can be done in about 11 cycles. By maintaining a separate PC and PSR for each context, a custom processor could switch contexts faster than our current implementation. However, Section 5 shows that even with 11 cycles of context switch overhead, multithreading significantly improves the system performance.

Sparcle's context switching mechanism can also be used to reduce synchronization overhead by switching contexts when a thread encounters a delay due to a synchronization variable access. This feature is not evaluated in this paper.

A multithreaded architecture is not free in terms of either its hardware or software requirements. The implementation of such an architecture requires multiple register sets or some other mechanism to allow fast context switches, additional network bandwidth, support logic in the cache controller, and extra complexity in the thread scheduling mechanism. Other methods for allowing multiple outstanding requests, such as weak ordering [1, 8, 10], incur similar implementation complexities in the cache controller. In Alewife, since the same context-switching mechanism is used for fast traps and for masking synchronization latencies, we feel the extra complexity in the processor is justified. See [2] for a detailed analysis of the interaction of multithreading with cache interference, network contention, context-switching overhead, and data-sharing effects.

## 4 The Simulation System

We use ASIM, the Alewife machine simulator, to estimate the extent to which multiple contexts can

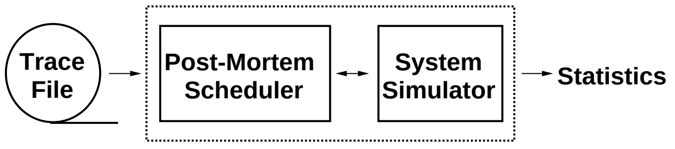

Figure 4: Coupled Trace-Driven Simulation.

overlap communication latency. ASIM is a cycle-by-cycle simulator of the entire machine, including processors, caches, and interconnection network. ASIM can use two sources of stimulus. First, programs written in a high-level language can be compiled, linked with a run-time system, and executed on the Alewife machine. Second, address traces of parallel programs containing embedded synchronization information can be used to drive the simulations. This paper uses the latter source.

ASIM uses a *coupled* trace-driven simulation method. In a coupled scheme, memory requests obey the synchronization constraints on the parallel program and respond to feedback from the interconnection network. This type of simulation models a correct ordering of memory requests. The coupled simulation system is depicted in Figure 4. An address trace of a single-processor execution of the parallel program drives the simulation system. The post-mortem scheduler generates a parallel trace, obeying the synchronization constraints specified in the uniprocessor trace. The scheduler sends address requests to the cache and network simulator, which responds to the requests from the scheduler and accumulates statistics that measure memory latency and other parameters.

The single processor trace is derived from a uniprocessor execution of an application parallelized using the single-processor-multiple-data (SPMD) computational model. Single processor traces are gathered using PSIMUL [18], a system for tracing parallel applications on IBM S/370 machines. In the SPMD model, each code section (task) in the system ends with a synchronization event, typically a barrier. The single processor trace describes the memory reference behavior of all the tasks and annotates the points where synchronizations must occur.

The dynamic post-mortem scheduler produces a parallel trace by simulating processors executing the task segments in the trace. Figure 5 illustrates its structure. Comparable simulators have been developed by other researchers [13]. The scheduler first makes a pass through the uniprocessor trace and constructs a task trace from the synchronization mark-

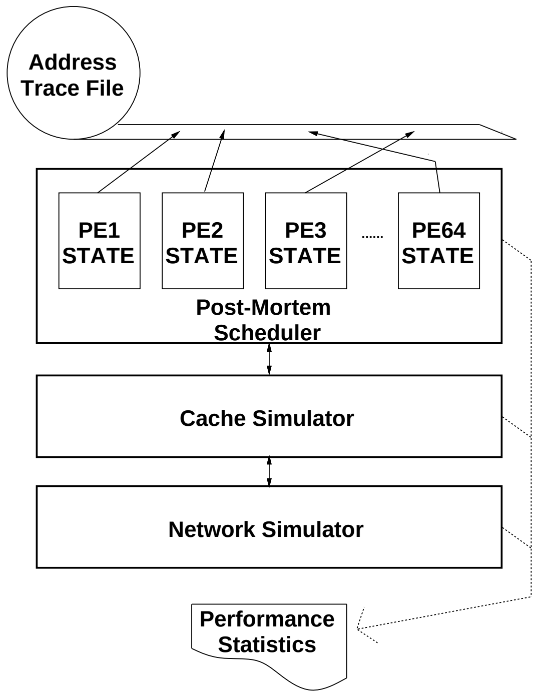

Figure 5: Coupled Post-mortem Scheduler.

ers. It then simulates the processors executing these tasks in a round-robin fashion with each processor making one reference each cycle from its task. The scheduler simulates the synchronization behavior of the processor and synthesizes synchronization references according to prespecified waiting algorithms. The implementation in this study uses software combining trees for barrier synchronizations [6].

In general, a processor issues a memory request from its task only after its previous network request is satisfied. However, the simulator can also choose to switch to a different thread after a network request to model a multithreaded processor. When simulating multithreaded processors, each processor maintains multiple control blocks corresponding to the state of multiple processor contexts.

The cache and network simulators model the memory system of the Alewife machine. Each processor has its own combined instruction and data cache. Each processing node also contains a portion of globally-shared memory. We assume that all instructions are replicated in each processor's local memory. Shared data is interleaved at a cache block size across all the processing nodes. We simulate several cache coherence protocols including full-map directories, LimitLESS directories, and chained directories. Since the effects of multithreading are uniform for all of the coherence protocols that we studied, we report results only for the LimitLESS protocol. The network is a two dimensional mesh.

| | SIMPLE | Weather |
|---|---|---|
| Total Memory Accesses | 24.0M | 26.7M |
| Instruction Fetch | 42.1% | 40.1% |
| Shared Read | 15.1% | 7.6% |
| Shared Write | 1.9% | 1.8% |
| Private Read | 26.7% | 42.8% |
| Private Write | 14.2% | 7.7% |
| CRCW PRAM Speed-up | 105 | 108 |

Table 1: Application Characteristics.

| | |
|---|---|
| Processing Elements | 64 |
| Coherence Protocol | LimitLESS$_4$ |
| Cache Size | 64KB (4096 lines) |
| Cache Block Size | 16 bytes |
| Network Topology | 2 Dim. Mesh ($8 \times 8$) |
| Channel Width | 16 bits |
| Network Speed | $2 \times$ processor speed |
| Memory Latency | 5 processor cycles |
| Context Switch | 11 processor cycles |

Table 2: Default Simulation Parameters.

## 5 Simulation Results

This section analyzes the performance of two FORTRAN applications when running on simulations of a 64 processor machine. After describing the parameters of the simulations, we compare the behavior of a multithreaded architecture to a standard configuration. We then analyze how multithreading affects the contribution of synchronization, local memory access latency, and remote memory access latency to the time that it takes to run each application. Finally, we compare multithreading with weak ordering, another method for masking remote memory access latency.

### 5.1 Simulation Parameters

Our simulations use two application programs to evaluate Alewife's multithreaded architecture. The SIMPLE application simulates the hydrodynamic and thermal behavior of a fluids in two dimensions, and the Weather application forecasts the state of the atmosphere given an initial state. Both SIMPLE and Weather are written in EPEX/FORTRAN [9, 20], a version of FORTRAN that has been extended with parallel constructs at IBM.

Table 1 summarizes the characteristics of these programs. The total number of memory references for each trace is the number of memory transactions that are issued in a uniprocessor execution of the trace. The table gives the percentage of each of five types of memory references in the trace. In a multiprocessor simulation, a processor's local memory module satisfies all cache misses on private data. Since the simulator interleaves shared data addresses over all of the memory modules in the system, cache misses on shared data are usually serviced by a remote memory module. The interleaved memory mapping policy differs from Alewife's memory system, which allocates a consecutive block of addresses to each processor's memory module. Interleaving is used for the simulations reported in this paper, because the post-mortem scheduler makes no attempt to address the problem of data allocation in a system with non-uniform memory access latencies.

We approximate the available parallelism of each application by simulating it on a 128 processor concurrent-read, concurrent-write parallel random access machine (CRCW PRAM). This model assumes that any memory transaction takes exactly one cycle to complete. The parallelism reported in Table 1 is the speed-up of each application on a CRCW PRAM.

Our metric for evaluating the performance of a multiprocessor is speed-up, the ratio between the execution time of an application on a uniprocessor, and the execution time on a multiprocessor. The uniprocessor execution time does not include any of the overhead inherent in multiprocessing, including synchronization and communication delay. Thus, the speed-up metric encapsulates all of the factors that contribute to a multiprocessor's execution time, including network latency, network contention, synchronization delay, cache coherence overhead, and load imbalance among processors.

In addition to determining the execution time of an application, the multiprocessor simulator generates raw statistics that measure an application's memory access patterns and the utilization of various system resources. We will use these statistics to explain the performance of the multithreaded architecture. The simulations reported in the following sections use the parameters listed in Table 2.

### 5.2 Effect of Multithreading

Figure 6 shows the simulated performance for the Weather and SIMPLE applications using one and two threads per processor. The vertical axis shows the speed-up of the application with 64 processors,

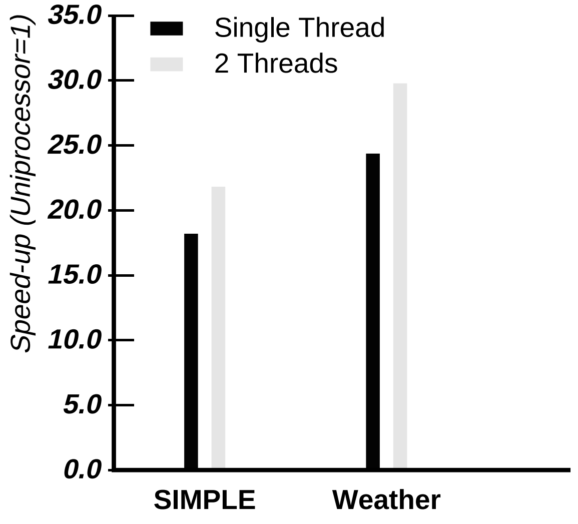

Figure 6: Effect of Multithreading.

relative to a uniprocessor execution of the same application without synchronization overhead. Both of the applications realize about a 20% performance increase from multithreading. Since neither of the application problem sets are large enough to sustain more than 128 contexts, no performance gain results from increasing the number of contexts from two to three per processor. This behavior is not surprising, given the fact that both applications' CRCW PRAM parallelism is less than 128. The simulations of several cache coherence protocols (including limited, full-map, and LimitLESS) show that the 20% performance increase is largely independent of the coherence mechanism. The performance of the single-link chain coherence protocol improved slightly more than the other protocols, due to the greater latency of requests to write shared data.

The simulations reveal the increase in the demand for network bandwidth due to multithreading. Table 3 shows the effect of multithreading on the average network channel utilization and the average packet transit latency. The higher bandwidth demands, evident from the increased channel utilizations, cause larger packet communication latencies. However, the speed-up results validate the prediction in [2]: as long as the network does not approach saturation, the benefits of tolerating remote access latency outweigh the increase in packet communication latency. Since the multithreaded architecture is designed to tolerate remote access latency, its higher network utilization indicates that it uses the system resources more efficiently than the standard (one thread) configuration. The efficient use of network and processor resources allows the multithreaded architecture to realize increased speed-up when running

| Application | Threads | Channel Utilization | Packet Latency |
|---|---|---|---|
| SIMPLE | 1 | 21.0% | 9.8 |
| | 2 | 25.7% | 12.3 |
| Weather | 1 | 14.7% | 8.4 |
| | 2 | 18.8% | 9.3 |

Table 3: Network Statistics.

the two applications.

The simulations performed during the Alewife design process will reveal the effects of multithreading on other applications. Some preliminary results are currently available from full system simulations run with compiled programs instead of trace-driven input. Applications that exhibit good communication locality did not perform better with multithreading, as expected. Other applications with poor locality, such as matrix transpose, exhibit 20% improvement with 2 contexts per processor and 25% with 4 contexts.

## 5.3 Cost Analysis

An analysis of the costs of memory transactions confirms the intuition that a multithreaded architecture yields better performance by reducing the effect of interprocessor communication latency. Our multiprocessor simulation system collects information about the distribution and the latency of various types of memory transactions. We have refined the statistics into a summary of the costs of four basic types of transactions.

1. *Application transactions* are the memory requests issued by the program running on the system. These transactions are the transactions in the original unscheduled trace.

2. *Synchronization transactions* are memory requests that implement the barrier executed at the end of a parallel segment of the application.

3. *Local cache miss transactions* occur when an application or synchronization transaction misses in the cache, but can be serviced in the local memory module.

4. *Remote transactions* occur when an application or synchronization transaction misses in the cache or requires a coherence action, resulting in a network transmission to a remote memory module. Multithreading is designed

to alleviate the latency caused by this type of transaction.

The contribution of each type of transaction to the time needed to run an application is equal to the number of transactions multiplied by the average latency of the transaction. We assume that the latency of application and synchronization transactions is equal to 1 cycle, while the simulator collects statistics that determine the average latency of the cache miss transactions. Table 4 shows the cost of each transaction type, normalized to the number of application transactions. For example, in the simulation of SIMPLE with one context per processor, the memory system spends an average of 3.98 cycles servicing remote transactions for every cycle it spends servicing an application data access.

The statistics in Table 4 are calculated directly from the raw statistics generated by the multiprocessor simulator using the parameters in Table 2, except for the cost of remote transactions in the multithreaded environment. A multithreaded architecture can overlap some of the cycles spent servicing remote transactions with useful work performed by switching to an active thread. We approximate the number of cycles that are overlapped from the average remote transaction latency, the context switch overhead, and the number of remote transactions. The number of overlapped cycles is subtracted from the latency of remote transactions in order to adjust the cost of remote transactions. For all of the simulations summarized in the table, the total cost multiplied by the number of application cycles is within 5% of the actual number of cycles needed to execute the application.

The analysis shows that remote transactions contribute a large percentage of the cost of running an application. This conclusion agrees with the premise that communication between processors significantly affects the speed of a multiprocessor. The multithreaded architecture realizes higher speed-up than the standard configuration, because it reduces the cost of remote transactions. Because communication latency grows with the number of processors in a system, the relative cost of remote transactions increases. This trend indicates that the effect of multithreading becomes more significant as system size increases.

## 5.4 Comparison to Weak Ordering

In order to evaluate the contribution of multithreading, we compare the performance of our multithreaded system to the performance of a system with weak ordering [1, 8, 10]. Weak ordering is an alternate method of tolerating remote memory access latency by allowing multiple outstanding transactions per processor. When using a weakly ordered memory system, if a programmer obeys certain synchronization semantics, then the system will appear to enforce sequential consistency, a convenient model of shared memory. In return, the memory system permits individual threads to overlap the latency of certain types of remote transactions with useful computation. In contrast, Alewife's memory system guarantees sequential consistency without placing restrictions on the synchronization semantics of programs, and overlaps remote transaction latency with multiple threads of computation.

To compare these two methods for masking remote memory latency, we instrumented the simulator to estimate the performance of a system with weak ordering. The simulator collects statistics about a weakly ordered memory system while simulating a single-context processor with a memory system that provides sequential consistency. For the purposes of our analysis, we modeled a weakly ordered system that allows all write transactions to be overlapped with computation, until the program encounters a synchronization point. At synchronization points, the system must wait for all write transactions to complete.

The approximation method records the latency of the memory transactions that may be overlapped with execution, namely write misses and attempts to write to read-only cache blocks. At any point in the simulation, the algorithm can determine the latest time that overlapped transactions will be completed for each processor. When a thread that is running on a processor performs a synchronization, the system knows how long to stall the processor to ensure that all of the outstanding memory transactions are complete. The simulator calculates the performance of weak ordering for each processor by subtracting overlapped cycles from the normal execution time and adding the extra wait time at synchronization points. The performance of weak ordering for the entire system is calculated by averaging the execution time over all processors.

Our simulations indicate that multithreaded ar-

| | SIMPLE | | Weather | |
|---|---|---|---|---|
| Transaction Type | 1 Thread | 2 Threads | 1 Thread | 2 Threads |
| Application | 1.00 | 1.00 | 1.00 | 1.00 |
| Synchronization | 1.17 | 1.08 | 0.76 | 0.45 |
| Local Cache Miss | 0.41 | 0.36 | 0.34 | 0.36 |
| Remote | 3.98 | 2.83 | 1.25 | 0.94 |
| Total | 6.56 | 5.27 | 3.35 | 2.75 |

Table 4: Memory access costs, normalized to application transactions.

| | Sequentially Consistent | | Weak |
|---|---|---|---|
| Application | 1 Thread | 2 Threads | Ordering |
| SIMPLE | 18.2 | 21.8 | 20.4 |
| Weather | 24.4 | 29.8 | 28.6 |

Table 5: Speed-up achieved by tolerating latency.

chitectures with two contexts compare favorably with weakly ordered systems. Table 5 shows the speed-up values that are measured for one thread (conventional), two thread, and weakly ordered systems. With the addition of a single extra thread, the multithreaded system achieves approximately the same speed up as the weakly ordered system. Both multithreading and weak ordering increase hardware and software complexity in order to provide latency tolerance. Both techniques use a similar approach to the latency problem, and both strategies result in comparable performance gains for our two applications.

## 6 Conclusions

This paper evaluated the potential for using a multithreaded architecture to tolerate remote memory access latencies in a shared memory multiprocessor. Simulations of a 64 processor system demonstrate performance gains with two contexts per processor and an 11 cycle context switch overhead. The higher performance of the multithreaded system indicates that the benefit of overlapping useful computation with communication latency outweighs the overhead of implementing multiple contexts per processor. For the applications that we studied, multithreaded processors realize a 20% performance increase over single-thread processors on a 64 processor Alewife machine.

Simulations of the Alewife system show that multithreading mitigates the negative impact of communication latency in multiprocessors. Because communication latency grows with the number of processors in a system, we expect the effect of multithreading to become even more significant as system size increases. Like any other technique that allows multiple outstanding memory requests, multithreading provides performance gains as long as the interconnection network has spare bandwidth.

Multithreading, a technique for tolerating remote access latency, should be viewed within the context of a layered approach to automatic management of communication locality in scalable multiprocessors. In a complete system, processor caches and compiler techniques decrease the proportion of memory accesses that require interprocessor communication. Run-time software can also attempt to minimize the average latency of remote accesses by efficiently allocating threads to the grid of processors. Multithreading provides the last line of defense when all of the attempts to exploit communication locality fail. The fundamental challenge in our research lies in understanding the interaction between the various mechanisms for implementing an efficient shared-memory system.

## 7 Acknowledgments

We are grateful the the other members of the Alewife development team, who have participated at every stage of the research reported in this paper. In particular, John Kubiatowicz is responsible for the design of the Alewife processing node, Beng-Hong Lim implemented the bulk of the run-time system that manages multithreading, and Gino Maa wrote the network simulator.

The serial version of the SIMPLE application was originally from Lawrence Livermore Laboratory. It was parallelized by Boris D. Lubachevsky while at NYU to run under WASHCLOTH, an instruction-level simulator of the NYU Ultracomputer, written by Allan Gottlieb. Pat Teller modified Lubachevsky's code to run under EPEX/FORTRAN at NYU. Weather was originally developed at the

NASA Space Flight Center. David Korn and Norman Rushfield parallelized the code to run under WASHCLOTH, and Pat Teller modified the code to run under EPEX/FORTRAN. Pat Teller and Allan Gottleib provided the sources for the programs. Harold Stone and Kimming So helped us obtain the traces from IBM. The static post-mortem scheduling process was developed by Mathews Cherian and Kimming So at IBM.

Machines used for simulations were donated by Sun Microsystems, Digital Equipment Corporation, and Encore Computer Corporation. The Sparcle implementation effort is a collaboration with LSI Logic Corporation and Sun Microsystems. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, and by grants from the Sloan Foundation and IBM.

## References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[2] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 1991. To appear.

[3] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[4] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[5] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

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

[7] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, ACM, April 1991. To appear.

[8] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 9–21, February 1988.

[9] D. A. George. *EPEX - Environment for Parallel Execution.* Technical Report RC 13158 (58851), IBM T. J. Watson Research Center, Yorktown Heights, September 1987.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

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

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

[13] Manoj Kumar and Kimming So. Trace Driven Simulation for Studying MIMD Parallel Computers. In *International Conference on Parallel Computing*, pages I–68 – I–72, 1989.

[14] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[15] G. M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

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

[17] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[18] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *PSIMUL - A System for Parallel Simulation of Parallel Systems.* Technical Report RC 11674 (58502), IBM T. J. Watson Research Center, Yorktown Heights, November 1987.

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

[20] J.M. Stone, F. Darema-Rogers, V.A. Norton, and G.F. Pfister. *Introduction to the VM/EPEX FORTRAN Preprocessor*. Technical Report RC 11407 (#51329), IBM T. J. Watson Research Center, Yorktown Height, NY, September 1985.

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

[22] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

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