The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor

Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz,
John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim,
Gino Maa, and Dan Nussbaum
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

The Alewife multiprocessor project focuses on the architecture and design of a large-scale parallel machine. The machine uses a low dimension direct interconnection network to provide scalable communication bandwidth, while allowing the exploitation of locality. Despite its distributed memory architecture, Alewife allows efficient shared memory programming through a multilayered approach to locality management. A new scalable cache coherence scheme called LimitLESS directories allows the use of caches for reducing communication latency and network bandwidth requirements. Alewife also employs run-time and compile-time methods for partitioning and placement of data and processes to enhance communication locality. While the above methods attempt to minimize communication latency, remote communication with distant processors cannot be completely avoided. Alewife's processor, Sparcle, is designed to tolerate these latencies by rapidly switching between threads of computation. This paper describes the Alewife architecture and concentrates on the novel hardware features of the machine including LimitLESS directories and the rapid context switching processor.

1 Introduction

High-performance computer design is driven by the need to solve computationally intensive problems efficiently and at a reasonable cost. While single-processor performance is limited by physical constraints, advances in technology make machines with thousands of processors feasible. Highly parallel machines offer significant cost-performance benefits over single processor machines.

Parallel machines are commonly organized as a set of nodes that communicate over an interconnection network, each node containing a processor and some memory. From the perspective of a node in a real machine built in three dimensional space, some nodes will be physically closer than others. Informally, a program running on a parallel machine displays *communication locality* (or memory reference locality) if the probability of communication (or access) to various nodes decreases with physical distance. Communication locality in parallel programs depends on the application as well as on partitioning and placement of data and processes.

Parallel machines are *scalable* if they can exploit communication locality in parallel programs. That is, for programs that display communication locality, scalable machines can offer proportionally better performance with more processing nodes [29]. Scalable machines are *easily programmable* if they provide automatic enhancement of communication locality in parallel programs.

The Alewife experiment explores methods for automatic enhancement of locality in a scalable parallel machine. The Alewife multiprocessor uses a distributed shared-memory architecture with a low-dimension direct network. Such networks are cost-effective, modular, and encourage the exploitation of locality [34, 19, 2]. Unfortunately, non-uniform communication latencies usually make such machines hard to program because the onus of managing locality invariably falls on the programmer. The goal of the Alewife project is to discover and to evaluate techniques for automatic locality management in scalable multiprocessors.

Alewife uses a multilayered approach to achieve this goal, consisting of techniques for *latency minimization* and *latency tolerance*. The compiler, runtime system, and hardware cooperate to enhance communication locality, thereby reducing average communication latency and required network bandwidth. Because remote communication with distant processors cannot always be avoided, Alewife's processor tolerates the resulting latencies by rapidly switching between threads of computation.

This paper focuses on the organization of the Alewife machine and describes its hardware features for automatic locality management. These features include shared-data caching, made possible by a new cache coherence scheme called LimitLESS directories, and rapid context switching. We present an overview of our approach to locality management in Section 2, and describe the machine organization and the programming environment in Section 3. Section 4 discusses the LimitLESS directory scheme, and Section 5 outlines our approach to latency tolerance. We also discuss the performance of the machine on a few applications. Other details of the machine are presented elsewhere [4, 8, 28]. Section 6 discusses related work, and Section 7 offers some perspective and summarizes the paper.

2 System Overview

The Alewife compiler, runtime system, and hardware try to reduce the communication latency where possible, and attempt to tolerate the latency otherwise. We are developing compiler technology to enhance the static communication locality of applications. Programs are first transformed into an intermediate task graph representation called WAIF [27], where the communication between threads is exposed through program analysis. Succeeding stages of the compiler map the task graph on to the machine and attempt to minimize overall execution time. When the compiler lacks enough information to make good placement decisions, it relegates the responsibility to the runtime layer.

Run-time software participates in enhancing locality through lazy task creation, a novel dynamic partitioning method [28], and intelligent scheduling. In a dynamic partitioning system the programmer or compiler can expose all of the parallelism in an application, but new tasks will be created at runtime only when there are idle processors. To enhance the likelihood of placing related tasks close to each other, a locality-based tree scheduler determines the order in which idle processors search for new tasks. To reduce the network bandwidth consumed by the searching processors, only single representatives from neighborhoods search for work. Simulations of several parallel applications with 64 processors showed that a mesh network yielded roughly the same speedup as a more expensive multistage network, when both used lazy task partitioning, a tree scheduler, and coherent caches.

Caching shared data is Alewife's hardware method for reducing memory access latency. With caches, the software does not need to worry as much about careful initial data placement; the caches dynamically move data objects close to the processor, so accesses are satisfied completely within a node. A new scalable scheme called *LimitLESS directories* solves the cache coherence problem. The LimitLESS directory is a small set of pointers (say 4) distributed along with each block of main memory that tracks copies of cached data and maintains memory consistency by transmitting invalidation messages over the network. The LimitLESS scheme allows a memory module to interrupt its local processor for software emulation of a full-map directory when the small set of pointers overflows. Section 4 describes and evaluates this scheme.

If the system cannot avoid a remote memory request, Alewife's processor can rapidly schedule another task in place of the stalled process. Alewife also tolerates synchronization latencies and provides fast traps through the *same* context switching mechanism. Because context switches are forced only on memory requests that require the use of the interconnection network and on synchronization faults, the processor achieves high single-thread performance.

We believe such a layered approach is necessary to build truly general-purpose parallel machines. Real applications are composed of phases, which will benefit in different proportions from the various layers. For example, matrix computations can benefit from static compiler analysis, while combinatorial search problems will profit from the runtime and cache layers. Finally, efficient execution of phases without inherent locality, such as matrix transpose, is possible when the processors can mask the latency of remote requests.

3 Hardware Organization of Alewife

Figure 1 depicts the Alewife machine as a set of processing nodes connected in a mesh topology. Each Alewife node consists of a processor, a cache, a portion of globally-shared distributed memory, a cache-memory-network controller, a floating-point coprocessor, and a network switch.

A single-chip controller on each node holds the cache tags and implements the cache coherence protocol by synthesizing messages to other nodes. While the Alewife architecture is scalable, the number of directory pointer bits in the current implementation of our controller will limit the maximum size of the machine to 512 nodes. The controller uses a simple message-based interface with the network. Various forms of shared memory coherence models are maintained by the controller via messages to other nodes. Alewife has a simple memory mapping scheme. The top few bits of the address determine the node number, and the rest of the address is the index within the specific module.

As shown in Figure 1, each node contains a network switch chip, specifically the Frontier series Mesh Routing Chip (FMRC) from Caltech. The mesh network uses wormhole routing [11] – a variant of cut-through routing [21]. The network has eight-bit channels, with a throughput of roughly 100M bytes per second in each direction. Free ports on peripheral nodes of the network are used for I/O, monitor, and host connections. The prototype Alewife system will attach to a host SUN backplane by interfacing a network switch to the VME bus.

The processor uses a *memory-reference-based interface* with the controller, although the controller uses a *message-based interface* for internode communications. Using a control word associated with each memory reference, various types of synchronization or communication types are synthesized by the processor. This interface allows a simple implementation of the processor.

Sparcle, a first-round prototype based on modifications to LSI Logic's SPARC processor [36] implementation, will clock at 33 MHz and context switch in 11 cycles. Each node has 64K bytes of direct-mapped cache and 4M bytes of globally-shared main memory. Each node has and an additional 4M bytes of

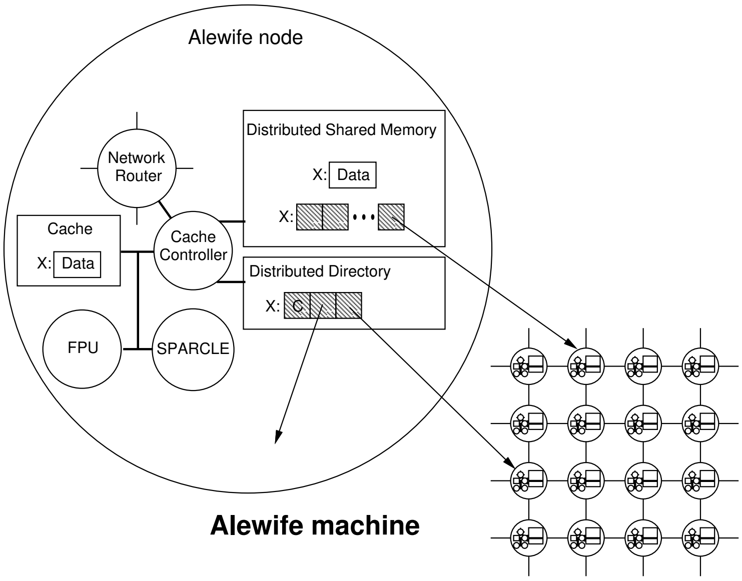

Figure 1: The Alewife machine, showing the LimitLESS directory extension.

local memory, a portion of which is used for the coherence directory. Alewife's cache and floating-point units are SPARC compatible. Sparcle uses a block multithreaded architecture [4].

Initially, our software system will be based on Mul-T [23]. A parallel C-like language is also under development. Mul-T's basic mechanism for generating concurrent threads is the `future` construct. The expression `(future` X`)`, where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *placeholder* to hold eventually the value of X. When created, the future is in an *unresolved* or *undetermined* state. When the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X and losing its identity as a future. Concurrency arises because the expression `(future` X`)` returns the future as its value without waiting for the future to resolve. Thus, the computation containing `(future` X`)` can proceed concurrently with the evaluation of X. The act of suspending computation if an object is an unresolved future and then proceeding when the future resolves is known as *touching* the object.

Our processor will allow operators to check for resolved futures with no overhead, disposing of the 60-100% overhead incurred by the system on other processors. Support for lightweight full-empty bit synchronization [35] in the processor will allow use of efficient fine-grain parallelism. In addition, the modified SPARC implementation is competitive in raw performance to contemporary sequential machines.

We propose to use Mul-T as our intermediate compiler language, augmented with primitives for specifying explicit partitioning and placement of both data and processes. Our compiler will partition a program taking communication costs into account, and produce an extended Mul-T program consisting of a set of tasks with granularity and placement information. The Orbit optimizing compiler [13, 22] will then compile these tasks to Sparcle machine code.

The design of the Alewife machine is in progress and a detailed simulator called ASIM is operational. ASIM implements several cache coherence protocols and interconnection network architectures. When ASIM is configured with its full statistics-gathering capability, it runs at about 5000 processor cycles per second on an unloaded SPARCserver 330. At this rate, a 64 processor machine runs approximately 80 cycles per second. Most of the simulations that we chose for this paper run for roughly one million cycles (a fraction of a second on a real machine), which takes 3.5 hours to complete. This lack of simulation speed is one of the primary reasons for implementing the Alewife machine in hardware — to enable a thorough evaluation of our ideas on much larger applications.

4 LimitLESS Directories

Shared data caching is an important component of Alewife's multilayered system for automatic locality management. Caches reduce the volume of traffic imposed on the network by providing demand-driven data replication where needed. However, replicating blocks of data in multiple caches introduces the cache coherence problem [15, 38]. A number of cache coherence protocols have been proposed to solve the coherence problem in network-based multiprocessors [6, 37, 5, 20]. These message-based protocols allocate a section of the system's memory, called a directory, to store the locations and state of the cached copies of each data block. The protocols send messages with data requests or invalidation signals, and record the acknowledgment of each of these messages to ensure global consistency of memory.

Although directory protocols have been around since the late 1970's, the usefulness of the early protocols (e.g., [6]) was in doubt for several reasons: First, the directory itself was a *centralized* monolithic resource which serialized all requests. Second, directory accesses were expected to consume a disproportionately large fraction of the available network bandwidth. Third, the directory became prohibitively large as the number of processors increased. To store pointers to blocks potentially cached by all the processors in the system, the size of the directory memory in early *full-map* protocols grows as $\Theta(N^2)$, where N is the number of processors in the system.

As observed in [5], the first two concerns are easily dispelled: The directory can be *distributed* along with main memory among the processing nodes to match the aggregate bandwidth of distributed main memory. Furthermore, required directory bandwidth is not much more than the memory bandwidth, because accesses destined to the directory alone comprise a small fraction of all network requests. Thus, the challenge lies in alleviating the severe memory requirements of the distributed full-map directory schemes.

Scalable coherence protocols differ in the size and the structure of the directory memory. *Limited directory* protocols [5], for example, avoid the severe memory overhead of full-map directories by allowing only a limited number of simultaneously cached copies of any individual block of data. Unlike a full-map directory, the size of a limited directory grows as $\Theta(N \log N)$ with the number of processors. Once all of the pointers in a directory entry are filled, the protocol must evict previously cached copies to satisfy new requests to read the data associated with the entry. In such systems, widely shared data locations degrade system performance by causing constant eviction and reassignment, or *thrashing*, of directory pointers. However, previous studies have shown that a small set of pointers is sufficient to capture the *worker-set* of processors that concurrently read many types of data [7, 39, 30]. The worker-set of a memory block is defined as the set of processors that concurrently read a memory location, and corresponds to the number of active pointers it would have in a full-map directory entry.

4.1 Overview of the LimitLESS Protocol

Alewife implements the LimitLESS cache coherence protocol, which nearly realizes the performance of the full-map directory protocol, with the memory overhead of a limited directory, but without excessive sensitivity to widely shared data. The LimitLESS scheme implements a small set of pointers in the memory modules, as do limited directory protocols. But when necessary, the scheme allows a memory module to interrupt the processor for software emulation of a full-map directory. Its name reflects the above properties: *Limit*ed directory *L*ocally *E*xtended through *S*oftware *S*upport.

Figure 1 depicts a set of directory pointers that correspond to the shared data block X, copies of which exist in several caches. In the figure, the software has extended the directory pointer array (which is shaded) into local memory.

The structure of the Alewife machine provides for an efficient implementation of this memory system extension. Since each processing node in Alewife contains both a memory controller and a processor, it is a straightforward modification of the architecture to couple the responsibilities of these two functional units, using the Sparcle processor's fast trap mechanism.

The LimitLESS scheme should not be confused with schemes previously termed

Component	Name	Meaning
Memory	Read-Only	Some caches have read-only copies of the data.
	Read-Write	Exactly one cache has a read-write copy.
	Read-Transaction	Holding read request, update is in progress.
	Write-Transaction	Holding write request, invalidation is in progress.
Cache	Invalid	Cache block may not be read or written.
	Read-Only	Cache block may be read, but not written.
	Read-Write	Cache block may be read or written.

Table 1: Directory states.

software-based, which require static identification of non-cacheable locations. Although the LimitLESS scheme is partially implemented in software, it dynamically detects when coherence actions are required; consequently, the software emulation should be considered a logical extension of the hardware functionality. To clarify the difference between protocols, schemes may be classified by function as *static* (compiler-dependent) or *dynamic* (using run-time information), and by implementation as *software-based* or *hardware-based*.

4.2 Protocol Specification

We now describe the LimitLESS directory protocol and the architectural interfaces needed to implement it.

The LimitLESS protocol has the same state transition diagram as the full-map protocol. The memory controller side of this protocol is illustrated in Figure 2, which contains the memory states listed in Table 1. These states are mirrored by the state of the block in the caches, also listed in Table 1. The state transition diagram specifies the states, the composition of the pointer set (P), and the transitions between the states. It is the responsibility of the protocol to keep the states of the memory and the cache blocks coherent. The protocol enforces coherence by transmitting messages between the cache/memory controllers. Every message contains the address of a memory block, to indicate which directory entry should be used when processing the message.

For example, Transition 2 from the Read-Only state to the Read-Write state is taken when cache i requests write permission (Write Request) and the pointer set is empty or contains just cache i ($P = \{\}$ or $P = \{i\}$). In this case, the pointer set is modified to contain i (if necessary) and the memory controller issues a message containing the data of the block to be written (Write Data).

Following the notation in [5], both full-map and LimitLESS are members of the $Dir_N NB$ class of cache coherence protocols. Therefore, from the point of view of the protocol specification, the LimitLESS scheme does not differ substantially from the full-map protocol. In fact, the LimitLESS protocol is also specified in

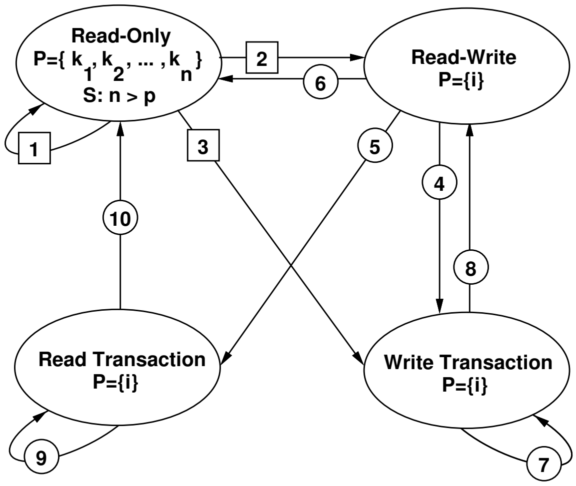

Figure 2: Directory state transition diagram.

Figure 2. The extra notation on the Read-Only ellipse ($S : n > p$) indicates that the state is handled in software when the size of the pointer set (n) is greater than the size of the limited directory (p). (See [8] for details). In this situation, the transitions with the square labels (1, 2, and 3) are executed by the interrupt handler on the processor that is local to the overflowing directory. When the protocol changes from a software-handled state to a hardware-handled state, the processor must modify the directory state so that the memory controller can resume responsibility for the protocol transitions.

4.3 Interfaces for LimitLESS

This section outlines the architectural features and hardware interfaces needed to support the LimitLESS directory scheme. To support the LimitLESS protocol efficiently, a cache-based multiprocessor needs several properties. First, it must be capable of rapid trap handling. Sparcle permits execution of trap code within five to ten cycles from the time a trap is initiated.

Second, the processor needs complete access to coherence related controller state such as pointers and state bits in the hardware directories. Similarly the directory controller must be able to invoke processor trap handlers when necessary. The hardware interface between the Alewife processor and controller, depicted in Figure 3, is designed to meet these requirements. The address and data buses permit processor manipulation of controller state and initiation of actions via load and store instructions to memory-mapped I/O space. In Alewife, the directories are placed in this special region of memory distinguished

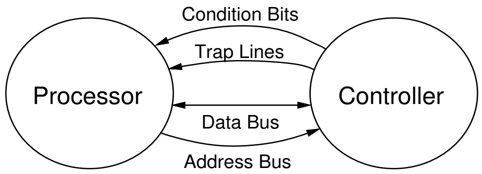

Figure 3: Signals between processor and controller.

from normal memory space by a distinct Alternate Space Indicator (ASI). The controller returns two condition bits and several trap lines to the processor.

Finally, a machine implementing the LimitLESS scheme needs an interface to the network that allows the processor to launch and to intercept coherence protocol packets. Although most shared-memory multiprocessors export little or no network functionality to the processor, Alewife provides the processor with direct network access through the Interprocessor-Interrupt (IPI) mechanism.

The Alewife machine supports a complete interface to the interconnection network. This interface provides the processor with a superset of the network functionality needed by the cache-coherence hardware. Not only can it be used to send and receive cache protocol packets, but it can also be used to send preemptive messages to remote processors (as in message-passing machines), hence the name Interprocessor-Interrupt.

We stress that the IPI interface is a single generic mechanism for network access – *not* a conglomeration of different mechanisms. The power of such a mechanism lies in its generality.

The current implementation of the LimitLESS trap handler is as follows: when an overflow trap occurs for the first time on a given memory line, the trap code allocates a full-map bit-vector in local memory. This vector is entered into a hash table. All hardware pointers are emptied and the corresponding bits are set in this vector. The directory state for that block is tagged Trap-On-Write. Emptying the hardware pointers allows the controller to continue handling read requests until the next pointer array overflow and maximizes the number of transactions serviced in hardware. However, the memory controller must interrupt the processor upon a write request. When additional overflow traps occur, the trap code locates the full-map vector in the hash table, empties the hardware pointers, and sets the appropriate bits in the vector.

Software handling of a memory line terminates when the processor traps on an incoming write request or local write fault. The trap handler finds the full-map bit vector and empties the hardware pointers as above. Next, it records the identity of the requester in the directory, sets the acknowledgment counter to the number of bits in the vector that are set, and places the directory in its normal Write Transaction state. Finally, it sends invalidations to all caches with bits set in the vector. The vector may now be freed. At this point, the memory line has returned to hardware control. When all invalidations are acknowledged, the hardware will send the data with write permission to the requester.

4.4 Performance Measurements

This section presents some preliminary results from the Alewife system simulator, comparing the performance of limited, LimitLESS, and full-map directories. The protocols are evaluated in terms of the total number of cycles needed to execute an application on a 64 processor Alewife machine. Using execution cycles as a metric emphasizes the bottom line of multiprocessor design: how fast a system can run a program.

The results presented below are derived from complete Alewife machine simulations and from dynamic post-mortem scheduler simulations. The complete-machine simulator runs programs written in the Mul-T language, optimized by the Mul-T compiler, and linked with a runtime system that implements both static work distribution and dynamic task partitioning and scheduling. Post-mortem scheduling, on the other hand, generates a parallel trace from a uniprocessor execution trace that has embedded synchronization information [9]. The post-mortem scheduler was implemented by Mathews Cherian with Kimming So at IBM. The post-mortem scheduler has been modified to incorporate feedback from the network in issuing trace requests [25].

To evaluate the benefits of the LimitLESS coherence scheme, we implemented an approximation of the new protocol in ASIM. During the simulations, ASIM simulates an ordinary full-map protocol, but when the simulator encounters a pointer array overflow, it stalls both the memory controller and the processor that would handle the LimitLESS interrupt for T_s cycles. The current implementation of the LimitLESS software trap handlers in Alewife suggests $T_s \approx 50$.

Table 2 shows the simulated performance of four applications, using a four-pointer limited protocol (Dir_4NB), a full-map protocol, and a LimitLESS (LimitLESS_4) scheme with $T_s = 50$. All of the runs simulate a 64-node Alewife machine with 64K byte caches and a two-dimensional mesh network.

Multigrid is a statically scheduled relaxation program, Weather forecasts the state of the atmosphere given an initial state, SIMPLE simulates the hydrodynamic and thermal behavior of fluids, and Matexpr performs several multiplications and additions of various sized matrices. The computations in Matexpr are partitioned and scheduled by a compiler. Weather and SIMPLE are measured using dynamic post-mortem scheduling of traces, while Multigrid and Matexpr are run on complete-machine simulations.

Since the LimitLESS scheme implements a full-fledged limited directory in hard-

Application	Dir_4NB	LimitLESS$_4$	Full-Map
Multigrid	0.729	0.704	0.665
SIMPLE	3.579	2.902	2.553
Matexpr	1.296	0.317	0.171
Weather	1.356	0.654	0.621

Table 2: Performance for three coherence schemes, in terms of millions of cycles.

ware, applications that perform well using a limited scheme also perform well using LimitLESS. Multigrid is such an application. All of the protocols require approximately the same time to complete the computation phase. This confirms the assumption that for applications with small worker-sets, such as multigrid, the limited (and therefore the LimitLESS) directory protocols perform almost as well as the full-map protocol. See [7] for more evidence of the general success of limited directory protocols.

To measure the performance of LimitLESS under extreme conditions, we simulated a version of SIMPLE with barrier synchronization implemented using a single lock (rather than a software combining tree). Although the worker-sets in SIMPLE are small for the most part, the globally shared barrier structure causes the performance of the limited directory protocol to suffer. In contrast, the LimitLESS scheme is less sensitive to wide-spread sharing.

The Matexpr application uses several variables that have worker-sets of up to 16 processors. Due to these large worker-sets time with the LimitLESS scheme is almost double that with the full-map protocol. The limited protocol, however, exhibits a much higher sensitivity to the large worker-sets.

Although software combining trees distribute barrier synchronization variables in Weather, one variable is initialized by one processor and then read by all of the other processors. Consequently the limited directory scheme suffers form hot-spot access to this location. As is evident from Table 2, the LimitLESS protocol avoids the sensitivity displayed by limited directories.

5 Using Multithreading to Tolerate Latency

While dynamic data relocation through caches reduces the average memory access latency, a fraction of memory transactions require service from remote memory modules. When transactions cause the cache coherence protocol to issue invalidation messages, the remote memory access latency is especially high. If 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. Processor idle time also results from synchronization delays.

One solution allows the processor to have multiple outstanding remote memory accesses or synchronization requests. Alewife implements this solution by using a processor that can rapidly switch between multiple threads of computation, and a cache controller that supports multiple outstanding requests. The controller forces a context switch when a thread issues a remote transaction or suffers an unsuccessful synchronization attempt. Processors that rapidly switch between multiple threads of computation are called *multithreaded architectures*.

The prototypical multithreaded architecture is the HEP [35]. In the HEP, the processor switches every cycle between eight processor-resident threads. Cycle-by-cycle interleaving of threads is also used in other designs [31, 18]. Such architectures are termed *finely multithreaded*. Although fine multithreading offers the potential of high processor utilization, it results in relatively poor scalar performance observed by any single thread, when there is not enough parallelism to fill all of the hardware contexts.

In contrast, Alewife employs *block multithreading* or coarse multithreading – context switches occur only when a thread executes a memory request that must be serviced by a remote node in the multiprocessor. Context switches are also forced when a thread encounters a delay due to a synchronization variable access. Thus, as long as a thread's memory requests hit in the cache or can be serviced by a local memory module, the thread continues to execute. Block multithreading allows a single thread to benefit from the maximum performance of the processor.

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, such as weak ordering [12, 1, 26], incur similar implementation complexities in the cache controller to allow multiple outstanding requests. In Alewife, because the same context-switching mechanism is used for fast traps and for masking synchronization latencies as well, we feel the extra complexity is justified.

5.1 Implementing a Multithreaded Processor

This section describes the implementation of the Sparcle processor and evaluates its potential in masking communication latency. Alewife's processor is designed to meet several objectives: it must context switch rapidly; it must support fast trap dispatching; and it must provide fine-grain synchronization.

Alewife's block multithreaded processor uses multiple register sets to implement fast context switching. The same rapid switching mechanism coupled with widely-spaced trap vectors minimizes the delay between the trap signal and the execution of the trap code. The wide spacing between trap dispatch points allows inlining of common trap routines at the dispatch point. The processor supports word-level full-empty bit synchronization. On a synchronization fault, the trap handling routine can respond by:

1. *spinning* – immediately return from the trap and retry the trapping instruction.
2. *switch spinning* – context switch without unloading the trapped thread.
3. *blocking* – unload the thread.

Sparcle is based on the following modifications to the SPARC architecture.

- Register windows in the SPARC processor permit a simple implementation of block multithreading. A window is allocated to each thread. The current register window is altered via SPARC instructions (`SAVE` and `RESTORE`). To effect a context switch, the trap routine saves the Program Counter (PC) and Processor Status Register (PSR), flushes the pipeline, and sets the Frame Pointer (FP) to a new register window. [4] 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 even faster. We show that even with 11 cycles of overhead and four processor resident contexts, multithreading significantly improves the system performance. See [40] for additional evidence of the success of multithreaded processors.
- The effect of multiple hardware contexts in the SPARC floating-point unit is achieved by modifying floating-point instructions in a context dependent fashion as they are loaded into the FPU and by maintaining four different sets of condition bits. A modification of the SPARC processor will make the context window pointer available externally to allow insertion into the FPU instruction.
- Sparcle detects unresolved futures through *word-alignment* and *non-fixnum* traps.
- The SPARC definition includes the Alternate Space Indicator (ASI) feature that permits a simple implementation of the general interface with the controller. The ASI is available externally as an eight-bit field and is set by special SPARC load and store instructions (`LDA` and `STA`). By examining the processor's ASI bits during memory accesses, the controller can select between different load/store and synchronization behavior.

- Through use of the Memory Exception (MEXC) line on SPARC, it can invoke synchronous traps and rapid context switching. Sparcle adds multiple synchronous trap lines for rapid trap dispatch to common routines. The controller can suspend processor execution using the MHOLD line. Inter-processor interrupts are implemented via asynchronous traps.

5.2 Simulation Results and Analysis

We compare the behavior of a multithreaded architecture to a standard configuration, and analyze how synchronization, local memory access latency, and remote memory access latency contribute to the run time of each application. See [3] for additional analyses.

A thorough evaluation of multithreading will require a large parallel machine and a scheduler optimized for multithreaded multiprocessors. On the largest machines we can reasonably simulate (around 64 processors) and with our current scheduler, the scheduling cost of threads generally outweighs the benefits of latency tolerance. Furthermore, the locality enhancement afforded by our caches and the runtime system diminishes the effect of non-local communications. Indeed, multithreading is expected to be the last line of defense when locality enhancement has failed. However it is still possible to observe the benefits of multithreading for phases of applications with poor communication locality.

Our simulation results are derived from both post-mortem scheduled and full system simulation branches of ASIM. The post-mortem scheduled runs use traces of SIMPLE and Weather as described in Section 4.4 and the full system simulations represent a transpose phase for a 256×256 matrix. 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 our multithreaded architecture. The simulations reported in the following sections use the parameters listed in Table 3.

5.3 Effect of Multithreading

Table 4 shows the run times for the various applications using one and two threads per processor. SIMPLE and Weather 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. For the matrix transpose phase, we realize a performance gain of about 20% with 2 threads and 25% with four threads.

Number of Processing Elements	64
Cache Coherence Protocol	LimitLESS_4
Cache Size	64KB (4096 lines)
Cache Block Size	16 bytes
Network Topology	2 Dimensional Mesh (8×8)
Network Channel Width	16
Network Speed	$2 \times$ processor speed
Memory Latency	5 processor cycles
Context Switch Time	11 processor cycles

Table 3: Default Simulation Parameters

Application	Contexts	Time
SIMPLE	1	2440123
	2	2034963
Weather	1	1405536
	2	1150325
Transpose	1	172242
	2	141571
	4	129450

Table 4: Effect of Multithreading

5.4 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. We refine the simulator statistics into 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 memory operations 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.

	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 5: Memory access costs, normalized to application transactions.

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 5 shows the cost of each transaction type, normalized to the number of application transactions for SIMPLE and Weather. 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 5 are calculated directly from the raw statistics generated by the multiprocessor simulator, 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.

6 Related Work

A hardware approach to the automatic reduction of non-local references that has achieved wide success in small-scale shared-memory systems is the use of high-speed caches to hold local copies of data needed by the processor. The memory consistency problem can be solved effectively on bus-based machines [15, 38] by exploiting their broadcast capabilities, but buses are bandwidth limited. Hence most shared-memory machines that deal with more than 8 or 16 processors do not support caching of shared data [17, 14, 32, 24].

Some recent efforts propose to circumvent the bandwidth limitation through various arrangements of buses and networks [41, 16, 26, 10]. However, buses cannot keep pace with improving processor technologies, because they suffer from clocking speed limitations in multidrop transmission environments. The DASH architecture does not really require the bus broadcast capability; rather, it uses a full-map directory scheme to maintain cache consistency. In contrast, Alewife is exploring the use of the LimitLESS directory for cache coherence, where the directory memory requirements grow as $\Theta(N \log N)$ with machine size

Chained directory protocols [20] are scalable in terms of their memory requirements, but they suffer from high invalidation latencies, because invalidations must be transmitted serially down the links. It is possible to use a block multithreaded processor such as Sparcle to mask the latency, or by implementing some form of combining. Accordingly, we have observed that chaining scheme enjoys a larger relative benefit from multithreading than the LimitLESS scheme. Chained protocols also require additional traffic to prevent fragmentation of the linked lists when cache locations are replaced. Furthermore, chained directory protocols lack the LimitLESS protocol's ability to couple closely with a multiprocessor's software.

Although caches are successful in automatic locality management in many environments, they are not a panacea. Caches rely on a very simple heuristic to improve communication locality. On a memory request, caches retain a local copy of the datum in the hope that the processor will reuse it before some other processor attempts to write to the same location. Thus repeat requests are satisfied entirely within the node, and communication locality is enhanced because remote requests are avoided. Caching and the associated coherence algorithms can be viewed as a mechanism for replicating and migrating data objects close to where they are used. Unfortunately, the same locality management heuristic is ill-suited to programs with poor data reuse; attempts by the programmer or compiler to maximize the potential reuse of data will not benefit all applications. In such environments, the ability to enhance the communication locality of references that miss in the cache and the ability to tolerate latencies of non-local accesses are prerequisites for achieving scalability.

The Alewife effort is unique in its multilayered approach to locality management: the compiler, runtime system and caches share the responsibility of intelligent partitioning and placement of data and processes to maximize communication locality. The block multithreaded processors mitigate the effects of unavoidable remote communication with their ability to tolerate latency.

7 Perspective and Summary

The class of MIMD machines is composed mainly of shared memory multiprocessors and message passing multicomputers. In the past, machine realizations of shared memory multiprocessors corresponded closely with the shared-memory programming model. Although the network took many forms, such as buses and multistage networks, shared memory was uniformly accessible by all the processors, closely reflecting the programmer's viewpoint. It was relatively easy to write parallel programs for such machines because the uniform implementation of shared memory did not require careful placement of data and processes. However, it has become abundantly clear that such architectures do not scale to more than few tens of processors, because an efficient implementation of uniform memory access is infeasible due to physical constraints.

Message passing machines, on the other hand, were built to closely match physical constraints, and message passing was the computational model of choice. In this model, no attempt was made to provide uniform access to all of memory, rather, access was limited to local memory. Communication with remote nodes required the explicit use of messages. Because they allowed the exploitation of locality, the performance of such architectures scaled with the size of the machine for applications that displayed communication locality. Unfortunately, the onus of managing locality was relegated to the user. The programmer not only had to worry about partitioning and placing data and processes to minimize expensive message transmissions, but also had to overcome the limitations of the small amount of memory within a node.

Recent designs reflect an increased awareness of the importance of simultaneously exploiting locality and reducing programming difficulty. Accordingly, we see a confluence in MIMD machine architectures with the emergence of distributed shared-memory architectures that allow the exploitation of communication locality, and message passing architectures with global addressability. A major challenge in such designs is the management of communication locality.

Alewife is a distributed shared-memory architecture that allows the exploitation of locality through the use of mesh networks. Alewife's network interface is message oriented, while the processor interface with the rest of the system is memory reference oriented. Alewife's approach to locality management is multilayered, encompassing the compiler, the runtime system, and the hardware.

While a more general compiler system is being developed, we have been experimenting with applications with special structure. Prasanna [33] has developed a compiler for expressions of matrix operations and FFTs. The system exploits the known structure of such computations to derive near-optimal process partitions and schedules. The Matexpr program used in Section 4 was produced by this system. The speedups measured on ASIM with this system outstrips the performance of parallel programs written using traditional heuristics.

A runtime system for Alewife is operational. The system implements dynamic process partitioning and near-neighbor tree scheduling. The tree scheduler currently uses the simple heuristic that threads closely related through their control flow are highly likely to communicate with each other. For many applications written in a functional style with the use of `futures` for synchronization the assumption is largely true, and the performance is superior to that of a randomized scheduler.

Caches are useful in enhancing locality for applications where there is a significant amount of reuse (assuming locality is related to the frequency and distance of remote communications). The LimitLESS directory scheme solves the cache coherence problem in Alewife. This scheme is scalable in terms of its directory memory use, and its performance is close to that of a full-map directory scheme.

The performance gap between LimitLESS and full-map is expected to become even smaller as the machine scales in size. Although in a 64-node machine, the software handling cost of LimitLESS traps is of the same order as the remote transaction latency of hardware-handled requests (about 50 cycles), the internode communication latency in much larger systems will be much more significant than the processors' interrupt handling latency. Furthermore, improving processor technology will make the software handling cost even less significant. If both processor speeds and multiprocessor sizes increase, handling cache coherence completely in software will become a viable option. Indeed, the LimitLESS protocol is the first step on the migration path towards interrupt-driven cache coherence.

Latency tolerance through the use of block multithreaded processors is Alewife's last line of defense when the other layers of the system are unable to minimize the latency of memory requests. The multithreaded scheme allows us to mask both memory and synchronization delays. The hardware support needed for block multithreaded also makes trap handling efficient.

The design of Alewife is in progress and a detailed simulator called ASIM is operational. The Sparcle processor has been designed; its implementation through modifications to an existing LSI Logic SPARC processor is in progress. A significant portion of the software system, including the dynamic partitioning scheme and the tree scheduler, is implemented and runs on ASIM. The Alewife compiler currently accepts hand partitioning and placement of data and threads; ongoing work focuses on automating the partitioning and placement. Several applications have been written, compiled, and executed on our simulation system.

8 Acknowledgments

The research reported in this paper is funded by NSF grant # MIP-9012773, DARPA contract # N00014-87-K-0825, and by grants from the Sloan foundation and IBM. Sparcle was implemented by Godfrey D'Souza (LSI Logic) and Mike Parkin (SUN Microsystems) by modifying LSI Logic's SPARC design. Sparcle's fabrication is being supported by LSI Logic. Generous equipment grants from SUN Microsystems, Digital Equipment Corporation, and Encore are also gratefully acknowledged.

References

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

[2] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 1991. To appear.

[3] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. September 1989. MIT VLSI Memo 89-566. Submitted for publication.

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

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

[6] 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.

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

[8] 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.

[9] Mathews Cherian. *A Study of Backoff Barrier Synchronization in Shared-Memory Multiprocessors.* S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[10] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. ParaDIGM: A Highly Scalable Shared-Memory Multi-computer Architecture. *IEEE Computer*. To appear.

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

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

[13] David A. Kranz et al. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of SIGPLAN '86, Symposium on Compiler Construction*, June 1986.

[14] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Saleh. Cedar – A Large Scale Multiprocessor. In *International Conference on Parallel Processing*, pages 524–529, August 1983.

[15] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, IEEE, New York, June 1983.

[16] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large Scale Cache-Coherent Multiprocessor. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 422–431, Hawaii, June 1988.

[17] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[18] 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.

[19] W. D. Hillis. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[20] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 74–77, June 1990.

[21] Parviz Kermani and Leonard Kleinrock. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks*, 3:267–286, October 1979.

[22] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. Technical Report YALEU/DCS/RR-632.

[23] David A. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, June 1989.

[24] James T. Kuehn and Burton J. Smith. The HORIZON Supercomputing System: Architecture and Software. In *Proc. Supercomputing '88*, November 1988.

[25] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors*. Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 49–58, June 1990.

[27] Gino Maa. The WAIF Intermediate Graphical Form. Oct. 1990. Alewife Memo.

[28] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990.

[29] Dan Nussbaum and Anant Agarwal. Scalability of Parallel Machines. *Communications of the ACM*, March 1991.

[30] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

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

[32] G. F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc. ICPP*, pages 764–771, August 1985.

[33] G. N. S. Prasanna. *Structure Driven Multiprocessor Compilation of Numeric Problems*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1990.

[34] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

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

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

[37] 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.

[38] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[39] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[40] 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.

[41] Andrew Wilson. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.