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

Anant Agarwal, David Chaiken, Godfrey D'Souza,[†] Kirk Johnson,
David Kranz, John Kubiatowicz, Kiyoshi Kurihara,[‡] Beng-Hong Lim,
Gino Maa, Dan Nussbaum, Mike Parkin,[§] and Donald Yeung
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-dimensional 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, 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, 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.

*An early version of this paper appears in the proceedings of the Workshop on Scalable Shared-Memory Multiprocessors, Seattle, June 1990, published by Kluwer Academic Publishers.

[†]Godfrey D'Souza is with LSI Logic.

[‡]Kiyoshi Kurihara is with IBM Japan, Ltd.

[§]Mike Parkin is with Sun Microsystems.

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 offer proportionally better performance with more processing nodes [28]. Scalable machines are *easily programmable* if they provide automatic enhancement of communication locality in parallel programs.

The Alewife project explores methods for automatic enhancement of locality in a scalable parallel machine. The *Alewife multiprocessor* uses a distributed shared-memory architecture with a low-dimensional direct network. Such networks are cost-effective, modular, and encourage the exploitation of locality [32, 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, run-time system, and hardware cooperate to enhance communication locality, thereby reducing average communication latency and required network bandwidth. However, high-latency communication cannot always be avoided. Alewife's processor tolerates the high latencies by switching rapidly between threads of computation.

This paper focuses on the organization of the Alewife machine and describes its hardware mechanisms for exploiting locality and for automatic locality management. These mechanisms include:

- a low-dimensional direct network;
- shared-data caching, made possible by a new cache-coherence scheme called LimitLESS directories, for improving communication locality during computations;
- rapid context switching, for tolerating unavoidable communication latencies, implemented in a modified commercial RISC processor called Sparcle.

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 concept of locality, and analyzes how reduced traffic rates and reduced communication distances resulting from communication locality translate to lower effective network latency. Section 5 discusses the LimitLESS directory scheme, and Section 6 outlines our approach to latency tolerance. Other details of the machine are presented elsewhere [3, 8, 26]. Section 7 discusses related work, and Section 8 offers some perspective and summarizes the paper.

2 System Overview

The Alewife compiler, run-time system, and hardware try to reduce communication latency where possible, and attempt to tolerate unavoidably long latencies. 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 [25], 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 run-time layer.

Run-time software participates in enhancing locality through lazy task creation, a novel dynamic partitioning method [26], and intelligent scheduling. In a dynamic partitioning system the programmer or compiler can expose all the parallelism in an application, but new tasks are created at run-time only when there are idle processors. To enhance the likelihood of placing related tasks close to each other, a locality based scheduler determines the order in which idle processors search for new tasks. In addition, the system provides annotation facilities that allow compiler-generated or programmer-specified information to be passed to the run-time system. These annotations facilitate more intelligent run-time partitioning, placement, and scheduling decisions.

Alewife's hardware reduces memory access latency by caching shared data. 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 coherence scheme called *LimitLESS directories* solves the cache-coherence problem. The LimitLESS directory uses a small set of pointers (say 4) distributed along with each block of main memory to track 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 5 describes and evaluates this scheme.

If the system cannot avoid a remote memory request, Alewife's processor can rapidly switch to another task using a fast-context-switching mechanism. Alewife also tolerates synchronization latencies and provides fast traps through the *same* context-switching mechanism. The processor achieves high single-thread performance because context switches are forced only on remote memory requests and synchronization faults.

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

3 Machine Organization and Programming

This section describes Alewife's hardware organization, the programming languages currently supported by the system, and the simulation environment, ASIM.

3.1 Hardware Organization

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. The controller implements the LimitLESS coherence protocol, described in detail in Section 5. As shown in the figure, up to five pointers per block are maintained in the hardware directory memory; when more pointers are needed, the controller allows the processor to extend the directory into local memory. The controller uses a simple message-based interface with the network. Various forms of shared-memory coherence

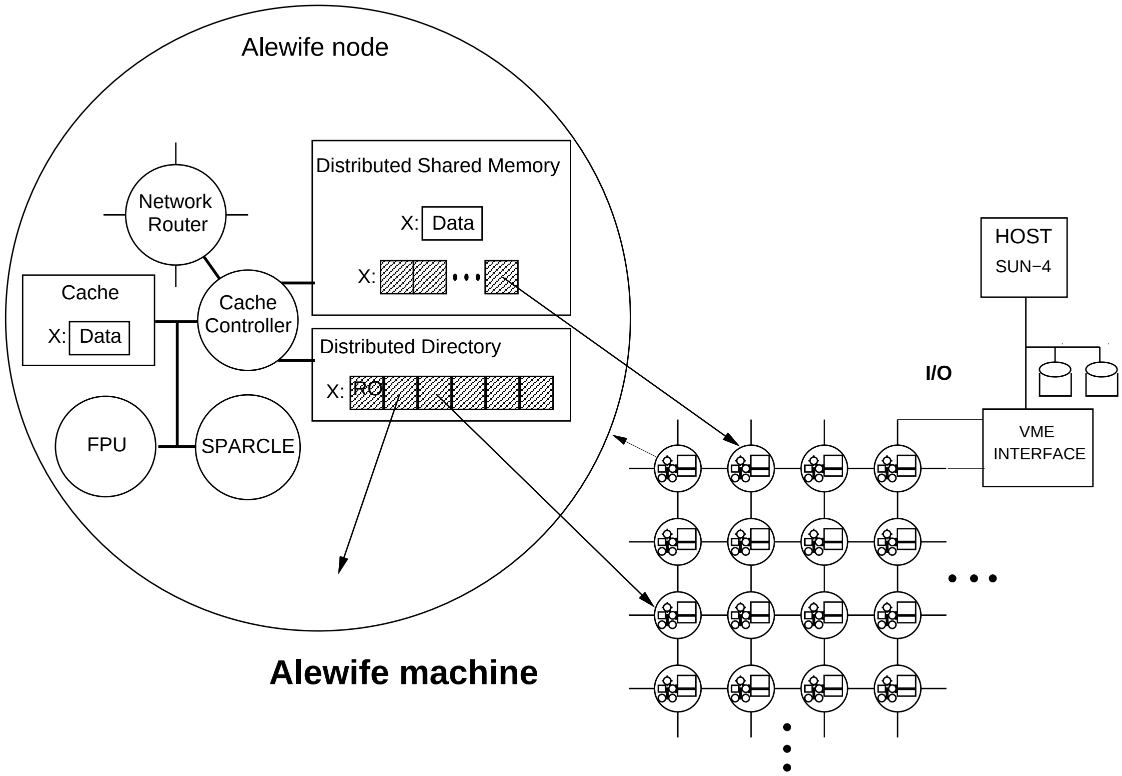

Figure 1: Structure of the Alewife machine.

models are maintained by the controller via messages to other nodes.

Each node has 64K bytes of direct-mapped cache and 4M bytes of globally-shared main memory. Each node has an additional 4M bytes of local memory, a portion of which is used for the coherence directory. The 32-bit address on SPARC therefore limits the maximum machine size to 512 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 node.

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. The network has eight-bit channels, and operates asynchronously with a switching delay of 50 nanoseconds per hop and a throughput of roughly 90M bytes per second in each direction. The controller chip on each node is responsible for synchronizing incoming data with the rest of the node. Message buffers within the switches are 19 flits deep. Messages are routed in the order of increasing dimension, which avoids deadlock in finite-buffered networks without end-around connections [10]. Deadlock, however, is still possible in a finite-buffered system, since the coherence protocol introduces cyclic dependencies between controllers. In Alewife, when the controller's network output queue is full for some period of time, the controller interrupts its processor. The processor then empties the input queue into local memory, thus simulating the effect of "infinite" buffers. 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 communicated by the processor to the controller. This interface allows a simple implementation of the processor.

Sparcle, a first-round prototype based on modifications to LSI Logic's SPARC processor implementation, will clock at around 33 MHz, and context switch in 14 cycles. Sparcle has been fully implemented and was taped out for fabrication to LSI Logic in September 1991. Alewife's cache and floating-point units are SPARC compatible. Sparcle uses a block-multithreaded architecture [3], details of which are presented in Section 6.

3.2 Support for Programming Languages

The Alewife system currently supports two programming languages: Mul-T [21], a parallel Lisp language, and Semi-C. Semi-C [20] is a parallel C-like language with extensions for expressing parallel execution. Semi-C supports most of the C language with the exception of pointer arithmetic and addressing operations. Excluding pointer arithmetic makes analysis of the code for parallel execution easier and allows the code to function in a garbage-collected environment. Both Mul-T and Semi-C support control-level and data-level parallelism.

Control-level parallelism may be expressed by wrapping `future` around an expression or statement X. The `future` keyword declares that X and the continuation of the future expression may be evaluated concurrently. The compiler or run-time system may choose to create a new task to evaluate X. In that case, an object known as a *placeholder* is also created and returned as the value of the future expression. The placeholder is created in an *undetermined* state. Any task that attempts to use the value of X before the evaluation of X has completed will encounter the undetermined placeholder and suspend until the value of X is available.

Data-level parallelism is expressed using parallel do-loops and fine-grain data-level synchronization is expressed by using J-structure and L-structure arrays. A J-structure is a data structure for producer-consumer style synchronization. It is like an array, but each element has additional state: *full* or *empty*. The initial state of a J-structure is empty. A reader of a J-structure element waits until the element's state is full before returning the value. A writer of a J-structure element writes a value, sets the state to full, and releases any waiting readers. An error is signalled if a write is attempted on a full element. To enable efficient memory allocation and good cache performance, J-structures are allowed to be reset to an empty state.

L-structures are similar to J-structures but support three operations: a locking read, a non-locking read, and a synchronizing write. A locking read waits until an element is full before emptying it (i.e., locking it) and returning the value. A non-locking read also waits until the element is full, but then returns the value without emptying the element. A synchronizing write stores a value to an empty element, and sets it to full, releasing any waiters. An L-structure therefore allows mutually exclusive access to each of its elements. In addition, L-structures allow multiple non-locking readers.

We use a slightly extended version of Mul-T as our intermediate compiler language, augmented with primitives for explicitly specifying partitioning and placement of both data and processes. Our compiler partitions 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 [22] compiles these tasks to Sparcle machine code.

The language features described are supported efficiently by the hardware. Placeholders for futures have a special tag (low bit set) that causes the Sparcle processor to trap when an arithmetic or pointer dereferencing instruction is attempted. This allows code to be generated as if no placeholders were present. Per-word, full-empty bits in memory [33], with support in Sparcle, allow efficient implementation of J-structures and L-structures as well as other types of fine-grain synchronization. In addition, the modified SPARC implementation of Sparcle is competitive in raw performance to contemporary sequential machines. More details are supplied in Section 6.

3.3 The Alewife Simulator

The implementation of Alewife is in progress and a detailed cycle-by-cycle simulator of the machine, ASIM [27], is operational. The software system, including compilers for Mul-T and Semi-C, and a run-time system, which supports both static and dynamic partitioning and placement of tasks and data, has been implemented and runs on ASIM. An alternate simulation platform for Alewife is Proteus, a multiprocessor simulator developed by Brewer and others at MIT [5]. While Proteus models Alewife less accurately than ASIM, it is roughly an order of magnitude faster, and can be customized to model other architectures.

ASIM includes modules for Sparcle, the floating-point coprocessor, the controller, and the network. In addition, ASIM implements several cache-coherence protocols and interconnection-network architectures, and allows a user to vary parameters such as number of processors, cache and memory sizes, network channel widths and buffer sizes, relative speeds of processors and network switches. ASIM can also draw its input from parallel address traces, correctly modeling synchronization behavior and feedback from the network using post-mortem scheduling. ASIM has proved invaluable not only for performance evaluations, but also for developing applications and systems software, and as a source for test vectors during the design verification of Sparcle and the controller.

ASIM non-intrusively gathers a large set of execution-level statistics; although this list is too large to include here, examples of statistics collected include parallelism profiles, communication-locality profiles, execution times, and synchronization wait-time distributions. When ASIM is configured with its full statistics-gathering capability, it runs at about 10,000 processor cycles per second on a SPARCstation II. At this rate, a 64-processor machine simulation runs at approximately 160 cycles per second. Most of the simulation results reported in this paper ran for a few million simulated cycles (a fraction of a second on a real machine), each of which took from several hours to a day 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 Communication Locality and Interconnection Networks

Alewife's distributed-memory architecture allows the exploitation of locality using a direct, mesh interconnection network. For programs that display communication locality, such networks offer good performance without the high cost of networks with higher dimensionality. Furthermore, for machines with a hundred to a thousand processors, the performance of these low-cost mesh networks is competitive with the performance of networks with higher dimensionality even when communication locality does not exist [2]. This section discusses the notion of communication locality and estimates the performance gains that result when networks exploit locality.

4.1 What is Locality?

Communication locality is a property of both applications (or algorithms) and parallel executions of programs. *Execution-level communication locality* exists if, during the execution of a parallel program, the frequency of communication with physically close processing nodes is higher than the frequency of communication with nodes further away. Thus, the average message distance is a good metric of execution-level locality.

Although this model applies straightforwardly to a message-passing style of computing, it applies equally well to a shared-memory computing style: View a memory access as a split-phase transaction, including a request message, some amount of work, and a response message. Some process — whether it is implemented in hardware or software — must be invoked to do the work necessary to handle the memory request. If a memory transaction can be satisfied by a cache or local memory, then the process is resident on the same node, and the communication distance of the request is zero. Other requests must travel some distance over the interconnection network to be serviced by a remote process. Using this notion of process-data duality, the shared-memory style is not materially different from a message-passing style from the viewpoint of communication.

Application-level communication locality is a property of programs, which translates to a *potential* for execution-level communication locality. Execution-level communication locality will exist only if the network can exploit locality and if the system can successfully preserve the application-level locality. Although a precise definition of application-level communication locality remains an open research issue, intuitively, there are at least two properties of programs that provide the potential for execution-level communication locality:

1. *Physical locality:* Programs whose communication graphs are sparse and have low bisection widths tend to have better execution-level communication locality than programs with higher bisections. The same applies to processes that communicate through shared variables if we treat each shared variable (or a portion of a shared data structure) as a process represented as a distinct node in a communication graph.

   It is easier to locate frequently communicating processes close to each other when communications are clustered between small sets of processes than when communication between all processes is equally likely. Placing data objects referenced solely by one process on the node where that process is run represents the exploitation of physical locality (assuming, as before, process-data duality from the viewpoint of communication).

2. *Temporal Locality:* When processes request the same data blocks frequently, execution-level locality can result if the data block is replicated on the requester's node. The resulting accesses that are successfully satisfied by the replicated copy have a communication distance of zero.

We stress that application-level locality does not automatically lead to execution-level locality. For example, no amount of physical locality in the application can compensate for poor placement of data or processes. Similarly, execution-level locality will lead to better performance only when the architecture can exploit locality. For this reason, Alewife employs distributed memory and direct networks, which allow full utilization of locality.

4.2 Locality in Multigrid

Let us examine the amount of communication locality evident in a parallel, blocked, multigrid computation [17] to illustrate concretely the various concepts presented above. The blocked multigrid algorithm displays a significant amount of both physical and temporal forms of application-level locality.

In blocked multigrid's communication graph, at each relaxation level, processes communicate solely with near neighbors. This property is evidence of physical locality. The resulting bisection grows as $\theta(\sqrt{N})$, where N is the number of nodes in the communication graph. A one-to-one mapping of the process communication graph to a mesh-connected multiprocessor results in largely near-neighbor communication. Some non-near-neighbor communication is also expected as the computation proceeds to higher relaxation levels.

Blocked multigrid also displays temporal locality. Because each relaxation step comprises multiple iterations, the data values internal to each block (excluding the perimeter values) are read and written frequently. If a cache is employed, communication is only necessary for the first-time accesses of these data values at a relaxation level.

We shall now examine the execution-level locality present in an execution of the multigrid algorithm. Processes and data are carefully assigned to processors to fully translate application-level physical locality to execution-level communication locality. The measurements of execution-level locality are taken from ASIM.

Figure 2 shows histograms of communication distances for all memory references and Figure 3 shows corresponding histograms for references to shared-data resident in the heap. In Alewife, references to local memory are satisfied entirely within the node, while references to memory located on other nodes result in message requests over the network. In the graphs, communication distances of zero correspond to references satisfied within the node.

First, let us inspect the memory references generated by the processors before they are filtered by caches. The locality histograms for these references are denoted "No Caching" in the figures. It is evident from the figures that the multigrid execution displays a significant amount of communication locality – very few messages travel long distances, and a significant fraction of all references are satisfied entirely within the node. Interestingly, we observe from the "No Caching" histograms that shared heap accesses comprise most of the messages to other nodes.

Now, let us compare the communication locality profiles of messages generated for systems with coherent caches and systems without caches. The bars denoted "With Caching, Misses" represent the locality profile of communications generated by references that are not satisfied by the cache, and the bars denoted "With Caching, Hits" represent the accesses that are satisfied by the cache (note, like cache misses to local memory, cache hits correspond to a distance of zero network hops). It is clear from the figures that the inclusion of caches improves the communication locality significantly. Over 90% of the heap traffic spread over a wide range of distances when caches were absent, gets lumped into the zero-distance column when caches are present. Put another way, the message rate drops by an order of magnitude. As discussed earlier, by replicating data where it is used, caches transform temporal locality in the application to communication locality in the execution.

Shared-data caching, unfortunately, introduces the cache-coherence problem. Section 5 describes Alewife's solution to this problem. The rest of this section analyses the expected performance gains from Alewife's mesh network when execution-level locality exists. The next section describes Alewife's techniques for translating application-level locality to execution-level locality.

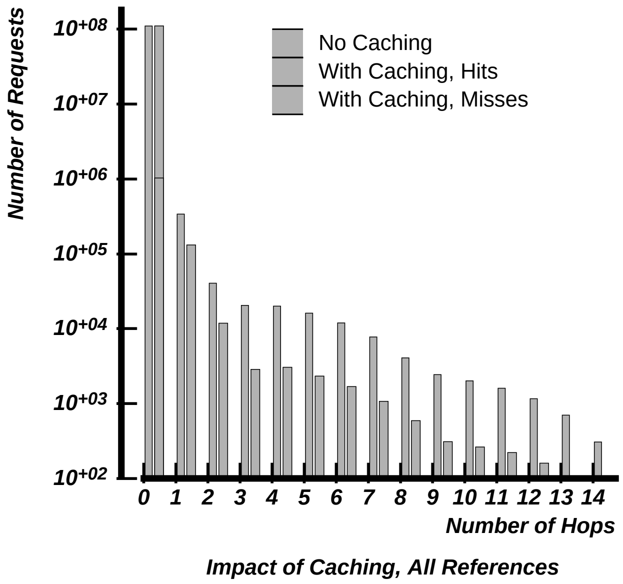

Figure 2: Communication locality profile for parallel multigrid computation. Histogram of message distances resulting from *all memory references*, before they are filtered by caches ("No Caching"), and in a system with coherence caches ("With Caching, Hits" and "With Caching, Misses"), for relaxing 128×128 grids on 64 processors with one context per processor.

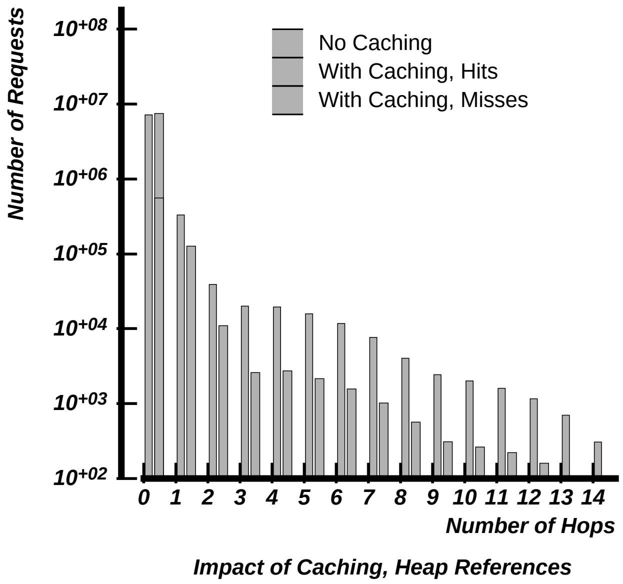

Figure 3: Communication locality profile for parallel multigrid computation. Histogram of message distances resulting from *heap references* only, before they are filtered by caches ("No Caching"), and in a system with coherence caches ("With Caching, Hits" and "With Caching, Misses"), for relaxing 128×128 grids on 64 processors with one context per processor.

4.3 Performance Benefits of Locality

Machine architectures can exploit communication locality in applications for improved performance when they provide mechanisms such as direct interconnection networks and memory (or caches) local to processors. In the rest of this section, using analytical models, we estimate the expected performance gains due to locality in direct, mesh interconnection networks. The performance of buffered, k-ary n-cube interconnection networks under traffic patterns that display communication locality is accurately captured by the following equation [2]:

$$T = \left[1 + \frac{\rho B}{(1-\rho)} \frac{(k_d - 1)}{k_d^2} \left(1 + \frac{1}{n}\right)\right] n k_d + B \qquad (1)$$

where T is the message latency, ρ is the network channel utilization, k_d is the average distance a message travels in each dimension of the network, and B is the message size. Assuming the network has unidirectional channels and end-around connections,[1] number of dimensions n, number of nodes in each dimension k, and message probability m,

$$\rho = mBk_d \qquad (2)$$

When message destinations are randomly chosen over the whole machine, that is, when the messages injected into the network display no locality, the average distance traveled in a dimension is given by $k_d = (k-1)/2$.

From our earlier discussion, communication locality tends to impact both the message request probability (m) and the distance (k_d) messages travel in each dimension. When communication distances can be reduced to zero, m is reduced, and when communicating processes can be placed physically close to each other, k_d decreases. Figure 4 plots network latency for various values of k_d and m. It is clear that reducing either m or k_d has a dramatic impact on network latency.

Locality improves latency because it reduces both the number of hops per packet and average contention delays. At light loads ($m \ll 1$), Equation 1 suggests that the contention component (containing the ρ term) can be ignored and that latency is linearly related to k_d. This linear relationship is clearly visible in Figure 4 for $m = 0.001$. The curve for $m = 0.015$ in the figure demonstrates that the impact of locality is much more significant when contention is high, because the latency at high loads is proportional to $1/(1 - mBk_d)$.

5 LimitLESS Directories

Shared-data caching is an important component of Alewife's multilayered system for automatic locality management. As illustrated by Figures 2 and 3, caches reduce the volume of traffic imposed on the network by providing demand-driven data replication. However, replicating blocks of data in multiple caches introduces the cache-coherence problem [14]. A number of cache-coherence protocols have been proposed to solve the coherence problem in network-based multiprocessors (e.g., [6, 4, 19]). 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

[1] Alewife's network is slightly different in that it has no end-around connections and has separate channels in each direction. The assumptions simplify the analysis without qualitatively changing the results.

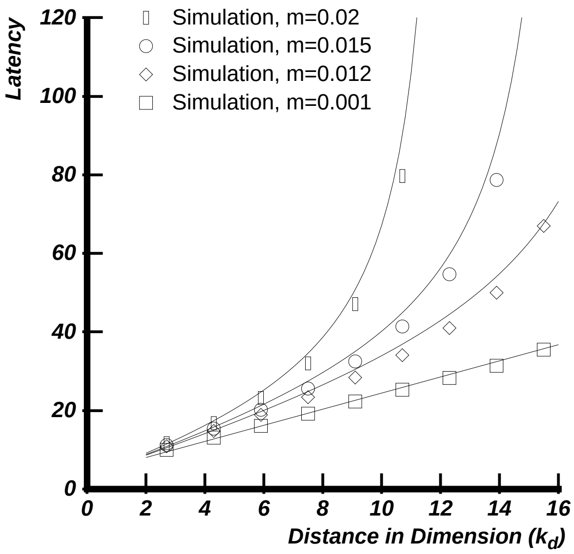

Figure 4: Impact of communication locality on network latency, with 1K processors, $n = 2$, $k = 32$, and $B = 4$. By our definition, communication locality reduces both the network request rate m and the effective communication distance k_d. Solid lines correspond to model predictions and points are taken from a simulator.

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., the full-map protocol in [6]) was in doubt for several reasons: First, the directory itself was a *centralized* monolithic resource that 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 [4], 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 [4], 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 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, 34, 29]. The worker-set of a memory block is defined as the set of processors that concurrently read a memory location, and corresponds to the active pointers the block would have in a full-map directory scheme.

5.1 Overview of the LimitLESS Protocol

Alewife implements the LimitLESS cache-coherence protocol, which realizes nearly 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 its local 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*, read-only 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 straightforward to couple the responsibilities of these two functional units using Sparcle's fast trap mechanism.

The LimitLESS scheme should not be confused with schemes usually termed software-based, which require static identification of non-cacheable locations. Although the LimitLESS scheme is partially implemented in software, it detects dynamically 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*.

5.2 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 side of the LimitLESS protocol is illustrated in Figure 5, 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. 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).

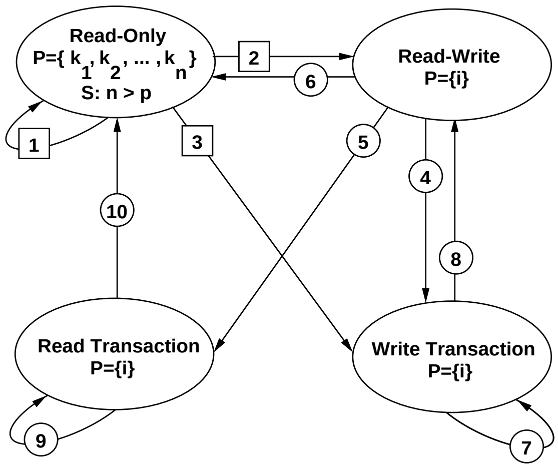

Figure 5: Directory state transition diagram.

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.

Following the notation in [4], both full-map and LimitLESS are members of the $Dir_N NB$ class of cache-coherence protocols. 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 5. 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, through the invocation of *overflow traps*. 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.

5.3 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 multiprocessor needs several properties. First, it must be capable of rapid trap handling. Sparcle permits the 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 6, is designed to meet these requirements. The address and data

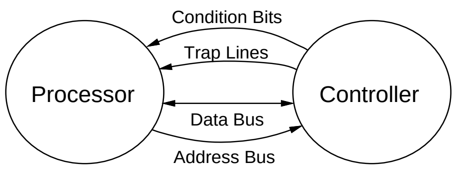

Figure 6: Signals between processor and controller.

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 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. While 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 IPI mechanism provides 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. 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 a directory 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. 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. When a write request occurs, the memory controller must interrupt the processor.

Software handling of a memory line terminates when the controller traps the processor on an incoming write request or local write fault. The trap handler finds the full-map bit vector and empties the hardware pointers as for a read request. Next, it records the identity of the write requester in the directory, and notes in an acknowledgment counter the number of bits in the vector that are set (which corresponds to the the number of caches with copies of the memory location). It then places the directory in the 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.

5.4 Performance Measurements

This section presents some results from the Alewife system simulator, ASIM, 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.

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: Application run times for three coherence schemes, in millions of cycles.

Using execution cycles as a metric emphasizes the bottom line of multiprocessor design: end performance.

To evaluate the benefits of the LimitLESS coherence scheme for a wide range of parameters, 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 $T_s \approx 50$. (While the LimitLESS trap code runs in about 200 cycles, these cycles are amortized over four read overflows as described above.)

Table 2 shows the simulated performance of three protocols: a four-pointer limited-directory protocol (Dir_4NB), a full-map protocol, and a four-pointer LimitLESS protocol (LimitLESS$_4$) with $T_s = 50$. All the runs simulate a 64-node Alewife machine with 64K-byte caches and a two-dimensional mesh network.

We use four applications in our simulations. Multigrid (which was discussed in Section 4.2) 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 hardware, applications that perform well using a limited scheme also perform well using LimitLESS. Multigrid is such an application. All the protocols require approximately the same time to complete. 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, the LimitLESS scheme takes twice as long as 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 the other processors. Consequently the limited-directory scheme suffers from hot-spot access to this location. As is evident from Table 2,

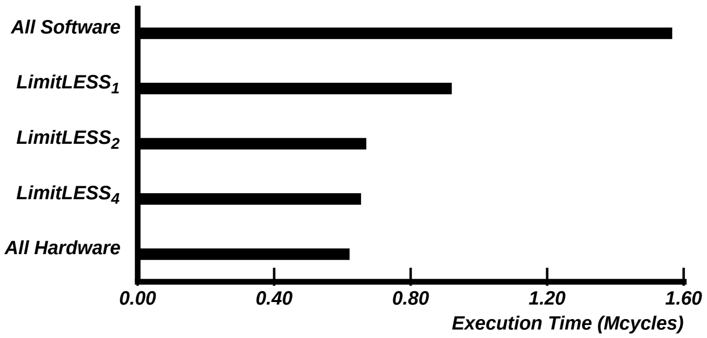

Figure 7: Performance of interrupt-driven all-software cache coherence for Weather.

the LimitLESS protocol avoids the sensitivity displayed by limited directories.

5.5 Towards Interrupt-Driven Cache Coherence

One of the most appealing feature of the LimitLESS protocol is its coupling of multiprocessor hardware and software, which attempts to make a tradeoff between performance and hardware cost. Software-handling of coherence transactions also allows run-time tuning of coherence policies. Furthermore, by understanding such hardware-software interfaces we believe it is possible to migrate increasing amounts of functionality into the software system without sacrificing performance as machines scale in size and processors get faster relative to communication speeds.

For example, would an interrupt-driven all-software cache-coherence scheme perform reasonably for Alewife using current technology? To investigate this question, we used ASIM to set the number of hardware pointers to zero, and simulated a processor trap on each coherence request. The resulting run times for Weather are shown in Figure 7. Remarkably, the all-software performance is only a factor of two worse than an all-hardware scheme. Since the Alewife cache controller is designed to allow experimenting with various numbers of hardware pointers, to a maximum of five, we will be able to conduct further experiments once we build the hardware.

The following simple model provides some intuition on why the performance gap between LimitLESS and full-map is small and why it will diminish as machines scale in size, allowing increased software participation in the coherence process. If T_h is the average remote-memory-access latency for a full-map directory protocol, T_s is the average delay for the full-map directory emulation interrupt (the software latency), and m_s is the fraction of memory accesses that overflow the small set of pointers implemented in hardware, then the average remote-memory-access latency for the LimitLESS protocol is approximated by

$$T_h \ + \ m_s T_s$$

Although T_h and T_s are comparable in a 64-processor Alewife machine, (for Weather, $T_h \approx 35$ cycles, $m_s \approx 3\%$, and $T_s \approx 50$ cycles), in much larger systems the internode communication latency will be much larger than the processors' interrupt handling latency ($T_h \gg T_s$). Furthermore, improving processor technology will make T_s even less significant compared to T_h, because a substantial fraction of T_h will arise due to wire delays. This approximation indicates that as processor speeds and multiprocessor sizes increase, servicing coherence transactions completely in software ($m_s = 1$) will become possible. In our experiment with Weather, the factor of 2.5 difference between $T_h + 1 * T_s$ and T_h explains the relative performance difference between all-software and hardware coherence. Thus, the reasoning behind the LimitLESS protocol indicates a trend towards interrupt-driven cache coherence.

6 Using Multithreading to Tolerate Latency

While dynamic data relocation using caches reduces the number of non-local memory references, some high-latency memory accesses will still occur. 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 can spend most of their time waiting for memory transactions to be serviced.

A similar problem arises when the processor must wait due to synchronizations. Synchronization latencies can be very high in large-scale multiprocessors resulting in large periods of processor idle time.

One solution that addresses *both* the problems of memory latencies and synchronization delays allows the processor to have multiple outstanding remote-memory accesses or synchronization requests. Alewife implements this solution by using a processor that can switch rapidly between multiple threads of computation and a cache controller that supports multiple outstanding requests.

When a thread issues a remote transaction or suffers an unsuccessful synchronization attempt, the Alewife controller traps the processor. If the trap resulted from a cache miss to a remote node, the trap handler forces a context switch. Otherwise, if the trap resulted from a synchronization fault, the trap handling routine chooses from one of four waiting mechanisms: [2]

1. *spinning* – immediately return from the trap and retry the trapping instruction.
2. *switch spinning* – context switch without unloading the trapped thread from the processor.
3. *blocking* – suspend the thread and unload it.
4. *switch blocking* – suspend the thread without unloading it, and switch to a different context.

Processors that switch rapidly between multiple threads of computation are called *multithreaded architectures*. Shared-resource multiprocessing was an earlier term used in this context by Flynn and Podvin [12]. The prototypical multithreaded machine is the HEP [33]. 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 [12, 30, 18]. Such architectures are termed *finely multithreaded*. Although fine multithreading offers the potential of high processor utilization, it results in relatively poor single-thread performance and low processor utilization when there is not enough parallelism to fill all the hardware contexts.

In contrast, Alewife employs *block multithreading* or coarse multithreading. That is, context switches occur only when a thread executes a memory request that must be serviced by a remote node in the multiprocessor, or on a failed synchronization request. Thus, the thread continues to execute as long as a thread's memory requests hit in the cache or can be serviced by a local memory module, and as long as synchronization attempts are successful. Block multithreading allows a single thread to benefit from the maximum performance of the processor.

[2] Specific algorithms for choosing among these mechanisms are described in [24].

A multithreaded architecture is not free: 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 [11, 1, 23], incur similar implementation complexities in the cache controller to allow multiple outstanding requests.

The appeal of block multithreading, however, lies in its generality: it is a single mechanism that allows tolerating read, write, and synchronization latencies. In Alewife, because the same context-switching mechanism is used for fast traps as well as for masking synchronization latencies, we feel the extra complexity is justified.

6.1 Implementing a Block-Multithreaded Processor

Sparcle is designed to meet several objectives: it must context switch rapidly, it must support fast trap dispatching, and it must provide efficient fine-grain synchronization. The extensive dependence on the trap mechanism to handle infrequent run-time situations, such as synchronization faults, cache misses to remote nodes, and network overflow, reflects an important aspect of Alewife's design discipline: provide hardware support to handle common cases efficiently, and rely on traps to detect rare cases and handle them in software.

Sparcle uses multiple register sets to implement fast context switching. The same rapid-switching mechanism minimizes the delay between the trap signal and the execution of the trap code.

To provide efficient fine-grained synchronization, the processor supports word-level full-empty bits. On a synchronization fault, the trap handling routine responds by selecting one of spinning, switch-spinning, blocking, or switch-blocking.

Sparcle is based on the following modifications to the SPARC architecture and its run-time software.

- 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 Current Window Pointer (CWP) to a new register window. [3] shows that even with a low-cost implementation, a context switch can be done in about 14 cycles. By maintaining a separate PC and PSR for each context, a custom processor could switch contexts even faster. Even with 14 cycles of overhead and four processor-resident contexts, multithreading significantly improves the system performance. See [35] for additional evidence of the success of multithreaded processors.

- The emulation 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 makes the frame pointer available externally to allow insertion into the FPU instruction.

- Sparcle detects unresolved futures through SPARC *word-alignment* and *tagged-arithmetic* traps, with the non-fixnum trap modified to look at only the low bit. The word-alignment and modified tagged-arithmetic traps automatically vector to a trap handler determined by the register number of the invalid operand.

- The SPARC architecture definition includes an *alternate space indicator* (ASI) feature that permits a simple implementation of a 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, the controller can invoke synchronous traps and rapid context switching. Sparcle adds multiple synchronous trap lines with distinct trap vectors for rapid trap dispatch to common routines. Inter-processor interrupts are implemented via asynchronous traps.

- The *window invalid mask* is used to implement switch blocking. The context-specific bits in the mask indicate whether the context is active or suspended. These bits can be written by the software, and new Sparcle instructions allow switching to the next active context.

- Spreading of trap vectors from four to 16 words: Since Alewife depends on fast trap processing, spreading the trap vectors permits time-critical trap code to reside entirely within the interrupt dispatch table.

6.2 Simulation Results and Analysis

In order to understand the benefits of block multithreading, compare the performance of two different applications on the Alewife architecture and on a single-thread configuration. For each application, we analyze how synchronization, local-memory-access latency, and remote-memory-access latency contribute to the run time of each application.

Observing the benefits of multithreading in small-scale simulations is difficult because the locality enhancement afforded by our caches and the run-time 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 a performance improvement due to multithreading in phases of applications with poor communication locality.

Our simulation results are derived from both post-mortem scheduled and full-system simulation in ASIM. The post-mortem scheduled runs use traces of SIMPLE and Weather as described in Section 5.4 and the full-system simulations represent a transpose phase for a 256×256 matrix. In addition to run times for each application, we present the breakdown of various communication costs and the utilization of different system resources. We will use these statistics to explain the performance of our multithreaded architecture. The simulations reported in the following sections use 64 processors and assume Alewife parameters (see Section 3).

6.3 Effect of Multithreading

Table 3 shows the run times for the various applications using one and two hardware contexts. 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 two threads and 25% with four threads.

Application	Contexts	Run Time
SIMPLE	1	2.440
	2	2.035
Weather	1	1.406
	2	1.150
Transpose	1	0.172
	2	0.142
	4	0.129

Table 3: Effect of Multithreading. Run times are in millions of cycles.

6.4 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 statistics gathered by the simulator to obtain 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 reduces the effect of synchronization and remote 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 4 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.

Table 4 approximates the remote transaction cost as follows. A multithreaded architecture can overlap some of the cycles spent servicing remote transactions with useful work performed by switching to an active thread. The number of overlapped cycles is subtracted from the latency of remote transactions in order to adjust the cost of remote transactions. For all 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

	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.

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.

7 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 [14] by exploiting their broadcast capabilities, but buses are bandwidth limited. Hence most shared-memory machines that deal with more than eight or 16 processors do not support caching of shared data [16, 13, 31].

Some recent efforts propose to circumvent the bandwidth limitation through various arrangements of buses and networks [36, 15, 9]. However, buses cannot keep pace with improving processor technologies, because they suffer from clocking-speed limitations in multidrop transmission environments. The Stanford DASH [23] architecture does not require the bus-broadcast capability; rather, it uses a full-map directory scheme to maintain cache consistency. The full-map's directory-memory size grows as $\Theta(N^2)$, where N is the number of processing nodes. 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. Additionally, the DASH implements a relaxed memory consistency model to tolerate memory latencies; Alewife achieves a similar result by implementing block multithreading.

Unlike full-map schemes, chained-directory protocols [19] are scalable in terms of their memory requirements, but they lack the LimitLESS protocol's ability to couple closely with a multiprocessor's software. They also suffer from high invalidation latencies, because invalidations must be transmitted serially down the links. It is possible to mask the latency by using a block-multithreaded processor such as Sparcle, 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.

Previous cache-coherent architectures relied largely on caches to relieve the communication-bandwidth bottleneck. 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, this 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, run-time 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.

8 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, such architectures do not scale to more than few hundreds 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 on such machines. In this model, no attempt was made to provide uniform access to all of memory; rather, access was limited to local memory. Communication between tasks required the explicit use of messages. Because such architectures allowed the exploitation of locality, their performance 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 locality.

Alewife is a distributed shared-memory architecture that allows the exploitation of locality through the use of direct 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 run-time system, and the hardware.

While a more general compiler system is being developed, we have been experimenting with applications with special structure. Prasanna 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 code used in Section 5 was produced by this system. The performance with this system outstrips the performance of programs written using traditional heuristics.

A run-time system for Alewife is operational. The system implements dynamic process partitioning and near-neighbor task 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.

Caches are useful in enhancing locality for applications that exhibit a significant amount of data 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. In a 64-node machine, the amortized software-handling cost of LimitLESS traps is of the same order as the remote-transaction-latency of hardware-handled requests. 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 avoid or minimize the latency of remote memory requests. Block multithreading allows us to mask both memory and synchronization delays. The hardware support needed for block multithreading also makes trap handling efficient.

The design of Alewife is in progress and a detailed simulator is operational. The Sparcle processor has been designed; its implementation through modifications to an existing LSI Logic SPARC was completed by LSI and SUN in March 1991, and MIT completed verification and testing of the design in September 1991. It is currently being fabricated by LSI Logic. A significant portion of the software system, including the dynamic-partitioning scheme and the tree scheduler are fully operational. 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.

9 Acknowledgments

We are grateful to Ravi Soundararajan, Marc Schaub, Terri Iuzzolino, Benson Wen, Craig Fields, Prasanna, Mathews Cherian, and Arthur Altman, for their contributions to the Alewife project. This research also benefited from discussions with Bill Dally, Mike Noakes, Tom Knight, and Steve Ward. J and L-structures were influenced by I-structures, developed by Arvind and others at MIT.

The research reported in this paper is funded in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-87-K-0825, in part by a NSF Presidential Young Investigator Award, and and in part by grants from the Sloan Foundation and IBM. IBM also made trace data available to us. 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. Chuck Seitz made the Mesh Routing Chips available to us. Pat Teller from NYU provided the SIMPLE and Weather programs. 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 *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[2] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[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, New York, June 1990. IEEE.

[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*, New York, June 1988. IEEE.

[5] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[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*, 23(6):41–58, 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)*, pages 224–234. ACM, April 1991.

[9] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. ParaDIGM: A Highly Scalable Shared-Memory Multi-computer Architecture. *IEEE Computer*, 1991. To appear.

[10] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *ieeetc*, c-36(5):547–553, May 1987.

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

[12] Michael J. Flynn and Albert Podvin. Shared resource multiprocessing. *IEEE Computer*, pages 20–28, March 1972.

[13] 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.

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

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

[16] 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.

[17] W. Hackbusch, editor. *Multigrid Methods and Applications*. Springer-Verlag, Berlin, 1985.

[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, New York, June 1988. IEEE.

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

[20] Kirk Johnson. Semi-C Reference Manual. ALEWIFE Memo No. 20, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1991.

[21] 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.

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

[23] 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, New York, June 1990. IEEE.

[24] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. Technical report, MIT VLSI Memo 91-632, February 1991.

[25] Gino Maa. The WAIF Intermediate Graphical Form. ALEWIFE Memo No. 23, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[26] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[27] Dan Nussbaum. ASIM Reference Manual. ALEWIFE Memo No. 13, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

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

[29] 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*, New York, June 1990. IEEE.

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

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

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

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

[34] 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.

[35] 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*, pages 273–280, New York, June 1989. IEEE.

[36] 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.