LimitLESS Directories: A Scalable Cache Coherence Scheme

David Chaiken, John Kubiatowicz, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Caches enhance the performance of multiprocessors by reducing network traffic and average memory access latency. However, cache-based systems must address the problem of cache coherence. We propose the LimitLESS directory protocol to solve this problem. The LimitLESS scheme uses a combination of hardware and software techniques to realize the performance of a full-map directory with the memory overhead of a limited directory. This protocol is supported by Alewife, a large-scale multiprocessor. We describe the architectural interfaces needed to implement the LimitLESS directory, and evaluate its performance through simulations of the Alewife machine.

1 Introduction

The communication bandwidth of interconnection networks is a critical resource in large-scale multiprocessors. This situation will remain unchanged in the future because physically constrained communication speeds cannot match the increasing bandwidth requirements of processors. Caches reduce the volume of traffic imposed on the network by automatically replicating data where it is needed. When a processor attempts to read or to write a unit of data, the system fetches the data from a remote memory module into a cache, which is a fast local memory dedicated to the processor. Subsequent accesses to the same data are satisfied within the local processing node, thereby avoiding repeat requests over the interconnection network.

In satisfying most memory requests, a cache increases the performance of the system in two ways: First, memory access latency incurred by the processors is shorter than in a system that does not cache data, because typical cache access times are much lower than interprocessor communication times (often, by several orders of magnitude). Second, when most requests are satisfied within processing nodes, the volume of network traffic is also lower.

However, replicating blocks of data in multiple caches introduces the cache coherence problem. When multiple processors maintain cached copies of a shared memory location, local modifications can result in a globally inconsistent view of memory. Buses in small-scale multiprocessors offer convenient solutions to the coherence problem that rely on system-wide broadcast mechanisms [4, 13, 15, 20, 24]. When any change is made to a data location, a broadcast is sent so that all of the caches in the system can either invalidate or update their local copy of the location. Unfortunately, this type of broadcast in large-scale multiprocessors negates the bandwidth reduction that makes caches attractive in the first place. In large-scale multiprocessors, broadcast mechanisms are either inefficient or prohibitively expensive to implement. Furthermore, it is difficult to make broadcasts atomic.

A number of cache coherence protocols have been proposed to solve the coherence problem in the absence of broadcast mechanisms [3, 6, 14, 23]. 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. Instead of broadcasting a modified location, the memory system sends an invalidate message to each cache that has a copy of the data. The protocol must also record the acknowledgment of each of these messages to ensure that the global view of memory is actually consistent.

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 early directory protocols (such as the Censier and Feautrier scheme [6]) allocate directory memory proportional to the product of the total memory size and the number of processors. While such *full-map* schemes permit unlimited caching, their directory size grows as $\Theta(N^2)$, where N is the number of processors in the system.

As observed in [3], 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.

Appeared in ASPLOS-IV. April, 1991.

Thus, recent research in scalable directory protocols focuses on 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 that is used to store the locations of cached blocks of data. *Limited directory* protocols [3], 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, because it allocates only a small, fixed number of pointers per entry. 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, 19, 25]. The performance of limited directory schemes can approach the performance of full-map schemes if the software is optimized to minimize the number of widely-shared objects.

This paper proposes the LimitLESS cache coherence protocol, which realizes the performance of the full-map directory protocol, with the memory overhead of a limited directory, but without excessive sensitivity to software optimization. This new protocol is supported by the architecture of the Alewife machine, a large-scale, distributed-memory multiprocessor. Each processing node in the Alewife machine contains a processor, a floating-point unit, a cache, and portions of the system's globally shared memory and directory. 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. Since this new coherence scheme is partially implemented in software, it can work closely with a multiprocessor's compiler and run-time system.

The LimitLESS scheme should not be confused with schemes previously termed 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*.

Chained directory protocols [14], another scalable alternative for cache coherence, avoid both the memory overhead of the full-map scheme and the thrashing problem of limited directories by distributing directory pointer information among the caches in the form of linked lists. But unlike the LimitLESS scheme, chained directories are forced to transmit invalidations sequentially through a linked-list structure (unless they implement some form of combining), and thus incur high write latencies for very large machines. Furthermore, chained directory protocols lack the LimitLESS protocol's ability to couple closely with a multiprocessor's software, as described in Section 6.

To evaluate the LimitLESS protocol, we have implemented the full-map directory, limited directory, and other cache coherence protocols in ASIM, the Alewife system simulator. Since ASIM is capable of simulating the entire Alewife machine, the different coherence schemes can be compared in terms of absolute execution time.

The next section describes the details of the Alewife machine's architecture that are relevant to the LimitLESS directory protocol. Section 3 introduces the LimitLESS protocol, and Section 4 presents the architectural interfaces needed to implement the new coherence scheme. Section 5 describes the Alewife system simulator and compares the different coherence schemes in terms of absolute execution time. Section 6 suggests extensions to the software component of the LimitLESS scheme, and Section 7 concludes the paper.

2 The Alewife Machine

Alewife is a large-scale multiprocessor with distributed shared memory. The machine, organized as shown in Figure 1, uses a cost-effective mesh network for communication. This type of architecture scales in terms of hardware cost and allows the exploitation of locality. Unfortunately, the non-uniform communication latencies 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, in order to insulate the programmer from the underlying machine details. Our approach to achieving this goal employs techniques for *latency minimization* and *latency tolerance*.

Alewife's multilayered approach, in which the compiler, runtime system, and hardware cooperate in enhancing communication locality, reduces average communication latency and required network bandwidth. Shared-data caching in Alewife is an example of a hardware method for reducing communication traffic. This method is dynamic, rather than static. Compiler partitioning and placement together with near-neighbor scheduling are Alewife's software methods for achieving the same effect.

When the system cannot avoid a remote memory request and is forced to incur the latency of the communication network, an Alewife processor rapidly schedules another process in place of the stalled process. Alewife can also tolerate synchronization latencies 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. Some systems have opted to use weak ordering [1, 11, 12] to tolerate certain types of communication latency, but this method lacks the ability to overlap read-miss and synchronization latencies. Although the Alewife cache coherence protocol enforces se-

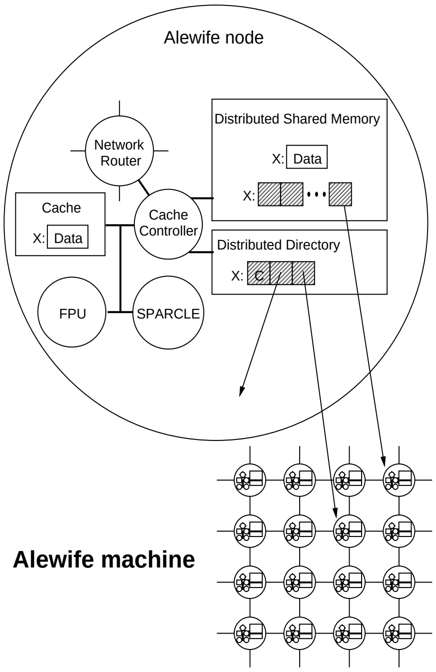

Figure 1: Alewife node, LimitLESS directory extension.

quential consistency [18], the LimitLESS directory scheme can also be used with a weakly-ordered memory model.

We have designed a new processor architecture that can rapidly switch between processes [2]. The first-round implementation of the processor called SPARCLE will switch between processes in 11 cycles. The rapid-switching features of SPARCLE allow an efficient implementation of LimitLESS directories.

An Alewife node consists of a 33 MHz SPARCLE processor, 64K bytes of direct-mapped cache, 4M bytes of globally-shared main memory, and a floating-point coprocessor. Both the cache and floating-point units are SPARC compatible [22]. The nodes communicate via messages through a direct network [21] with a mesh topology using wormhole routing [10]. A single-chip controller on each node holds the cache tags and implements the cache coherence protocol by synthesizing messages to other nodes. Figure 1 is an enlarged view of a node in the Alewife machine. Because the directory itself is distributed along with the main memory, its bandwidth scales with the number of processors in the system. The SPARCLE processor is being implemented jointly with LSI Logic and SUN Microsystems through modifications to an existing SPARC design. The design of the cache/memory controller is also in progress.

3 The LimitLESS Protocol

As do limited directory protocols, the LimitLESS directory scheme capitalizes on the observation that only a few shared memory data types are widely shared among processors. Many shared data structures have a small *worker-set*, which is defined as the set of processors that concurrently read a memory location. The worker-set of a memory block corresponds to the number of active pointers it would have in a full-map directory entry. The observation that worker-sets are often small has led some researchers to propose the use of a hardware cache of pointers to augment the limited-directory for a few widely-shared memory blocks [19]. However, when running properly optimized software, a directory entry overflow is an exceptional condition in the memory system. We propose to handle such "protocol exceptions" in software. This is the integrated systems approach — handling common cases in hardware and exceptional cases in software.

The LimitLESS scheme implements a small number of hardware pointers for each directory entry. If these pointers are not sufficient to store the locations of all of the cached copies of a given block of memory, then the memory module interrupts the local processor. The processor then emulates a full-map directory for the block of memory that caused the interrupt. 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. This scheme is called LimitLESS, to indicate that it employs a *Limit*ed directory that is *L*ocally *E*xtended through *S*oftware *S*upport. Figure 1 is an enlarged view of a node in the Alewife machine. The diagram 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.

Since Alewife's SPARCLE processor is designed with a fast trap mechanism, the overhead of the LimitLESS interrupt is not prohibitive. The emulation of a full-map directory in software prevents the LimitLESS protocol from exhibiting the sensitivity to software optimization that is exhibited by limited directory schemes. But given current technology, the delay needed to emulate a full-map directory completely in software is significant. Consequently, the LimitLESS protocol supports small worker-sets of processors in its limited directory entries, implemented in hardware.

3.1 A Simple Model of the Protocol

Before discussing the details of the new coherence scheme, it is instructive to examine a simple model of the relationship between the performance of a full-map directory and the LimitLESS directory scheme. Let T_h be the average remote memory access latency for a full-map directory protocol. T_h includes factors such as the delay in the cache and memory controllers, invalidation latencies, and network latency. Given T_h, the formula $T_h + mT_s$ approximates the average remote memory access latency for the LimitLESS protocol. T_s (the software latency) is the average delay for the full-map directory emulation interrupt, and m is the fraction of memory accesses that overflow the small set of pointers implemented in hardware.

For example, our simulations of a Weather trace on a 64 node Alewife system (see Section 5) indicate that $T_h \approx$

Type	Symbol	Name	Data?
Cache to Memory	RREQ	Read Request	
	WREQ	Write Request	
	REPM	Replace Modified	√
	UPDATE	Update	√
	ACKC	Invalidate Ack.	
Memory to Cache	RDATA	Read Data	√
	WDATA	Write Data	√
	INV	Invalidate	
	BUSY	Busy Signal	

Table 1: Protocol messages for hardware coherence.

35 cycles. If $T_s = 100$ cycles, then remote accesses with the LimitLESS scheme are 10% slower (on average) than with the full-map protocol when $m \approx 3\%$. In the Weather program, 97% of accesses hit in the limited directory.

LimitLESS directories are scalable, because the memory overhead grows as $\Theta(N \log N)$, and the performance approaches that of a full-map directory as system size increases. Although in a 64 processor machine, T_h and T_s are comparable, 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. This approximation indicates that if both processor speeds and multiprocessor sizes increase, handling cache coherence completely in software ($m = 1$) will become a viable option. The LimitLESS protocol is the first step on the migration path towards interrupt-driven cache coherence. Other systems [9] have experimented with handling cache misses entirely in software.

3.2 Protocol Specification

In the above discussion, we assume that the hardware latency (T_h) is approximately equal for the full-map and the LimitLESS directories, because 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 2. These states are mirrored by the state of the block in the caches, also listed in Table 2. 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 (listed in Table 1) 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. Table 1 also indicates whether a message contains the data associated with a memory block.

The state transition diagram in Figure 2 specifies the states, the composition of the pointer set (P), and the transitions between the states. Each transition is labeled with a number that refers to its specification in Table 3. This table annotates the transitions with the following information: 1. The *input message* from a cache that initiates the transaction and the identifier of the cache that sends it. 2. A *precondition* (if any) for executing the transition. 3. Any *directory entry change* that the transition may require. 4. The *output message* or messages that are sent in response to the input message. Note that certain transitions require the use of an acknowledgment counter (*AckCtr*), which is used to ensure that cached copies are invalidated before allowing a write transaction to be completed.

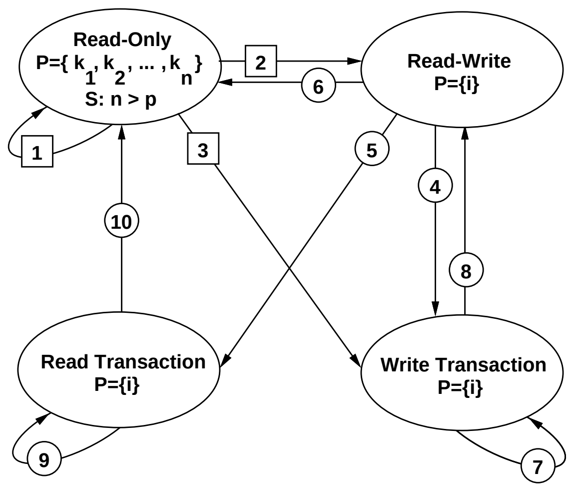

Figure 2: Directory state transition diagram.

For example, Transition 2 from the Read-Only state to the Read-Write state is taken when cache i requests write permission (WREQ) 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 (WDATA).

Following the notation in [3], 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. 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). 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.

The Alewife machine will support an optimization of the LimitLESS protocol that maximizes the number of transactions serviced in hardware. When the controller interrupts the processor due to a pointer array overflow, the processor completely empties the pointer array into local memory, allowing the controller to continue handling read requests until the next pointer array overflow. This optimization is called Trap-On-Write, because the memory controller must interrupt the processor upon a write request, even though it can handle read requests itself.

Component	Name	Meaning
Memory	Read-Only	Some number of caches have read-only copies of the data.
	Read-Write	Exactly one cache has a read-write copy of the data.
	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 2: Directory states.

Transition Label	Input Message	Precondition	Directory Entry Change	Output Message(s)
1	$i \rightarrow$ RREQ	—	$P = P \cup \{i\}$	RDATA $\rightarrow i$
2	$i \rightarrow$ WREQ	$P = \{i\}$	—	WDATA $\rightarrow i$
	$i \rightarrow$ WREQ	$P = \{\}$	$P = \{i\}$	WDATA $\rightarrow i$
3	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \notin P$	$P = \{i\}$, AckCtr $= n$	$\forall k_j$ INV $\rightarrow k_j$
	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \in P$	$P = \{i\}$, AckCtr $= n - 1$	$\forall k_j \neq i$ INV $\rightarrow k_j$
4	$j \rightarrow$ WREQ	$P = \{i\}$	$P = \{j\}$, AckCtr $= 1$	INV $\rightarrow i$
5	$j \rightarrow$ RREQ	$P = \{i\}$	$P = \{j\}$, AckCtr $= 1$	INV $\rightarrow i$
6	$i \rightarrow$ REPM	$P = \{i\}$	$P = \{\}$	—
7	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ ACKC	AckCtr $\neq 1$	AckCtr $=$ AckCtr $- 1$	—
	$j \rightarrow$ REPM	—	—	—
8	$j \rightarrow$ ACKC	AckCtr $= 1$, $P = \{i\}$	AckCtr $= 0$	WDATA $\rightarrow i$
	$j \rightarrow$ UPDATE	$P = \{i\}$	AckCtr $= 0$	WDATA $\rightarrow i$
9	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ REPM	—	—	—
10	$j \rightarrow$ UPDATE	$P = \{i\}$	AckCtr $= 0$	RDATA $\rightarrow i$
	$j \rightarrow$ ACKC	$P = \{i\}$	AckCtr $= 0$	RDATA $\rightarrow i$

Table 3: Annotation of the state transition diagram.

4 Interfaces for LimitLESS

This section discusses the architectural features and hardware interfaces needed to support the LimitLESS directory scheme. We describe how these interfaces are supported in the Alewife machine. Since the Alewife network interface is somewhat unique for shared-memory machines, it is examined in detail. Afterwards, we introduce the additional directory state that Alewife supports, over and above the state needed for a limited directory protocol, and examine its application to LimitLESS.

To support the LimitLESS protocol efficiently, a cache-based multiprocessor needs several properties. First, it must be capable of rapid trap handling, because LimitLESS is an extension of hardware through software. The Alewife machine couples a rapid context switching processor (SPARCLE) with a finely-tuned software trap architecture, permitting 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

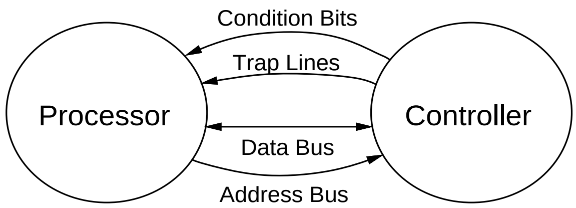

Figure 3: Signals between processor and controller.

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) [2]. 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 following subsections outline the implementations of the IPI mechanism, the LimitLESS directory modes, and the trap han-

Source Processor Packet Length Opcode
operand 0 operand 1 ⋮ operand $m-1$
data word 0 data word 1 ⋮ data word $n-1$

Figure 4: Uniform packet format for the Alewife machine.

dlers.

4.1 Interprocessor-Interrupt (IPI)

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.

Network Packet Structure To simplify the IPI interface, network packets have a single, uniform structure, shown in Figure 4. This figure includes only the information seen at the destination; routing information is stripped off by the network. The Packet Header contains the identifier of the source processor, the length of the packet, and an opcode. It is a single word in the Alewife machine. Following the header are zero or more operands and data words. The distinction between operands and data is software-imposed; however, it is a useful abstraction supported by the IPI interface.

Opcodes are divided into two distinct classes: protocol and interrupt. Protocol opcodes are used for cache-coherence traffic. While they are normally produced and consumed by the controller hardware, they can also be produced or consumed by the LimitLESS trap-handler. Protocol opcodes designate the type of coherence transaction. For example, a cache read miss generates a message with <opcode = RREQ>, <Packet Length = 2>, and <Operand0 = Address>. Packets with protocol opcodes are called protocol packets.

Interrupt opcodes have their most significant bit set and are used for interprocessor messages. Their format is defined entirely by the software. Packets with interrupt opcodes are called interprocessor interrupts and are processed in software at their destinations.

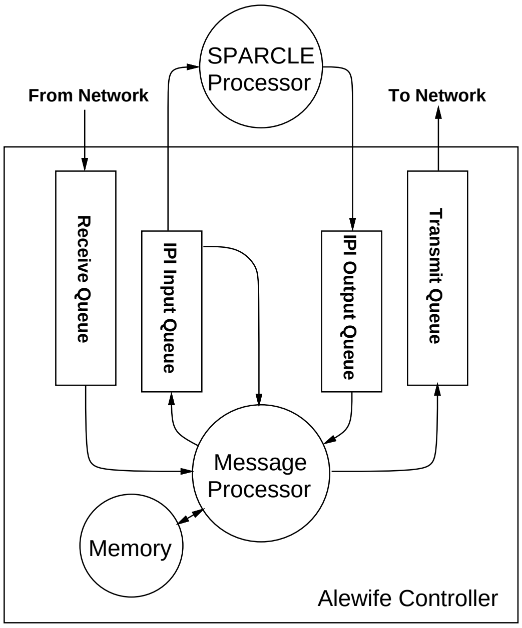

Figure 5: Queue-based diagram of the Alewife controller.

Transmission of IPI Packets A simplified, queue-based diagram of the Alewife cache/memory controller is shown in Figure 5. This is a "memory-side" diagram; for simplicity it excludes the processor cache.

The processor interface uses memory-mapped store instructions to specify destination, opcode, and operands. It also specifies a starting address and length for the data portion of the packet. Taken together, this information completely specifies an outgoing packet. Note that operands and data are distinguished by their specification: operands are written explicitly through the interface, while data is fetched from memory. The processor initiates transmission by storing to a special trigger location, which enqueues the request on the *IPI output queue*.

Reception of IPI Packets When the controller wishes to hand a packet to the processor, it places it in a special input buffer, the *IPI input queue*. This queue is large enough for several protocol packets and overflows into the network *receive queue*. The forwarding of packets to the IPI queue is accompanied by an interrupt to the processor.

The header (source, length, opcode) and operands of the packet at the head of the IPI input queue can be examined with simple load instructions. Once the trap routine has examined the header and operands, it can either discard the packet or store it to memory, beginning at a specified location. In the latter case, the data that is stored starts from a specified offset in the packet. This store-back capability permits message-passing and block-transfers in addition to enabling the processing of protocol packets with data.

Meta State	Description
Normal	Directory handled by hardware.
Trans-In-Progress	Interlock, software processing.
Trap-On-Write	Trap: WREQ, UPDATE, REPM.
Trap-Always	Trap: all incoming packets.

Table 4: Directory meta states for LimitLESS scheme.

IPI input traps are synchronous, that is, they are capable of interrupting instruction execution. This is necessary, because the queue topology shown in Figure 5 is otherwise subject to deadlock. If the processor pipeline is being held for a remote cache-fill [1] and the IPI input queue overflows, then the receive queue will be blocked, preventing the load or store from completing. At this point, a synchronous trap must be taken to empty the input queue. Since trap code is stored in local memory, it may be executed without network transactions.

4.2 Meta States

As described in Section 3, the LimitLESS scheme consists of a series of extensions to the basic limited directory protocol. That section specifies circumstances under which the memory controller would invoke the software. LimitLESS requires a small amount of hardware support in addition to the IPI interface.

This support consists of two components, *meta states* and *pointer overflow trapping*. Meta states are directory modes listed in Table 4. They may be described as follows:

- The hardware maintains coherence for Normal-mode memory blocks. The worker-sets of such blocks are no larger than the number of hardware pointers.
- The Trans-In-Progress mode is entered automatically when a protocol packet is passed to software (by placing it in the IPI input queue). It instructs the controller to block on *all* future protocol packets for the associated memory block. The mode is cleared by the LimitLESS trap code after processing the packet.
- For memory blocks in the Trap-On-Write mode, read requests are handled as usual, but write requests (WREQ), update packets (UPDATE), and replace-modified packets (REPM) are forwarded to the IPI input queue. When packets are forwarded to the IPI queue, the directory mode is changed to Trans-In-Progress.
- Trap-Always instructs the controller to pass all protocol packets to the processor. As with Trap-On-Write, the mode is switched to Trans-In-Progress when a packet is forwarded to the processor.

[1] While the Alewife machine switches contexts on remote cache misses (see [2]) under normal circumstances, certain forward-progress concerns dictate that we occasionally hold the processor while waiting for a cache-fill.

The two bits required to represent these states are stored in directory entries along with the states of Figure 2 and five hardware pointers.

Controller behavior for pointer overflow is straightforward: when a memory line is in the Read-Only state and all hardware pointers are in use, then an incoming read request for this line (RREQ) will be diverted into the IPI input queue and the directory mode will be switched to Trans-In-Progress.

Local Memory Faults What about local processor accesses? A processor access to local memory that must be handled by software causes a *memory fault*. The controller places the faulting address and access type (i.e. read or write) in special controller registers, then invokes a synchronous trap.

A trap handler must alter the directory when processing a memory fault to avoid an identical fault when the trap returns. To permit the extensions suggested in Section 6, the Alewife machine reserves a one bit pointer in each hardware directory entry, called the *Local Bit*. This bit ensures that local read requests will never overflow a directory. In addition, the trap handler can set this bit after a memory fault to permit the faulting access to complete.

4.3 LimitLESS Trap Handler

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 mode is set to Trap-On-Write before the trap returns. 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 (WREQ) 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 the Normal mode, 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.

Since the trap handler is part of the Alewife software system, many other implementations are possible.

5 Performance Measurements

This section describes some preliminary results from the Alewife system simulator, comparing the performance of

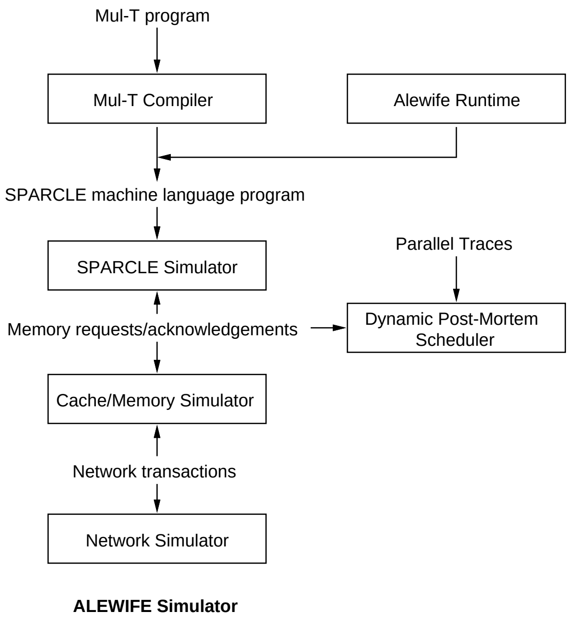

Figure 6: Diagram of ASIM, the Alewife system simulator.

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.

5.1 The Measurement Technique

The results presented below are derived from complete Alewife machine simulations and from dynamic post-mortem scheduler simulations. Figure 6 illustrates these two branches of ASIM, the Alewife Simulator.

ASIM models each component of the Alewife machine, from the multiprocessor software to the switches in the interconnection network. The complete-machine simulator runs programs that are written in the Mul-T language [16], optimized by the Mul-T compiler, and linked with a runtime system that implements both static work distribution and dynamic task partitioning and scheduling. The code generated by this process runs on a simulator consisting of processor, cache/memory, and network modules.

Although the memory accesses in ASIM are usually derived from applications running on the SPARCLE processor, ASIM can alternatively derive its input from a dynamic post-mortem trace scheduler, shown on the right side of Figure 6. Post-mortem scheduling is a technique that generates a parallel trace from a uniprocessor execution trace that has embedded synchronization information [8]. The post-mortem scheduler is coupled with the memory system simulator and incorporates feedback from the network in issuing trace requests, as described in [17]. The use of this input source is important because it lets us expand the workload set to include large parallel applications written in a variety of styles.

The simulation overhead for large machines forces a trade-off between application size and simulated system size. Programs with enough parallelism to execute well on a large machine take an inordinate time to simulate. 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.

To evaluate the potential benefits of the LimitLESS coherence scheme, we implemented an approximation of the new protocol in ASIM. The technique assumes that the overhead of the LimitLESS full-map emulation interrupt is approximately the same for all memory requests that overflow a directory entry's pointer array. This is the T_s parameter described in Section 3. 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. While this evaluation technique only approximates the actual behavior of the fully-operational LimitLESS scheme, it is a reasonable method for determining whether to expend the greater effort needed to implement the complete protocol.

5.2 Performance Results

Table 5 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 $T_s = 50$. The table presents the performance of each application/protocol combination in terms of the time needed to run the program, in millions of processor cycles. 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. The Weather and SIMPLE applications 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 of the protocols, including the four-pointer limited directory (Dir_4NB), the full-map directory, and the LimitLESS scheme require approximately the same time to complete the computation phase. This confirms the assumption that for applications

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

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 performs almost as well as the full-map directory protocol, because LimitLESS is able to distribute the barrier structure to as many processors as necessary.

The Matexpr application uses several variables that have worker-sets of up to 16 processors. Due to these large worker-sets, the processing 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.

Weather provides a case-study of an application that has not been completely optimized for limited directory protocols. Although the simulated application uses software combining trees to distribute its barrier synchronization variables, Weather has one variable initialized by one processor and then read by all of the other processors. Our simulations show that if this variable is flagged as read-only data, then a limited directory performs just as well for Weather as a full-map directory.

However, it is easy for a programmer to forget to perform such optimizations, and there are some situations where it is very difficult to avoid this type of sharing. Figure 7 gives the execution times for Weather when this variable is not optimized. The vertical axis on the graph displays several coherence schemes, and the horizontal axis shows the program's total execution time (in millions of cycles). The results show that when the worker-set of a single location in memory is much larger than the size of a limited directory, the whole system suffers from hot-spot access to this location. So, limited directory protocols are extremely sensitive to the size of a heavily-shared data block's worker-set.

The effect of the unoptimized variable in Weather was not evident in previous evaluations of directory-based cache coherence [7], because the network model did not account for hot-spot behavior. Since the program can be optimized to eliminate the hot-spot, the new results do not contradict the conclusion of [7] that system-level enhancements make large-scale cache-coherent multiprocessors viable. Nevertheless, the experience with the Weather application reinforces the belief that complete-machine simulations are necessary to evaluate the implementation of cache coherence.

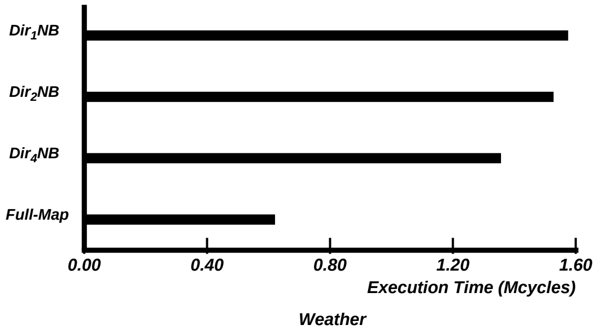

Figure 7: Limited and full-map directories.

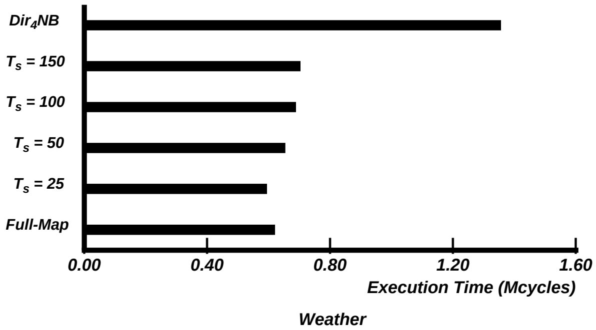

Figure 8: LimitLESS, 25 to 150 cycle emulation latencies.

As shown in Figure 8, the LimitLESS protocol avoids the sensitivity displayed by limited directories. This figure compares the performance of a full-map directory, a four-pointer limited directory (Dir_4NB), and the four-pointer LimitLESS (LimitLESS$_4$) protocol with several values for the additional latency required by the LimitLESS protocol's software (T_s = 25, 50, 100, and 150). The execution times show that the LimitLESS protocol performs about as well as the full-map directory protocol, even in a situation where a limited directory protocol does not perform well. Furthermore, while the LimitLESS protocol's software should be as efficient as possible, the performance of the LimitLESS protocol is not strongly dependent on the latency of the full-map directory emulation. The current implementation of the LimitLESS software trap handlers in Alewife suggests $T_s \approx 50$.

It is interesting to note that the LimitLESS protocol, with a 25 cycle emulation latency, actually performs better than the full-map directory. This anomalous result is caused by the participation of the processor in the coherence scheme. By interrupting and slowing down certain processors, the LimitLESS protocol produces a slight back-off effect that reduces contention.

The number of pointers that a LimitLESS protocol implements in hardware interacts with the worker-set size of data structures. Figure 9 compares the performance of Weather with a full-map directory, a limited directory, and Limit-

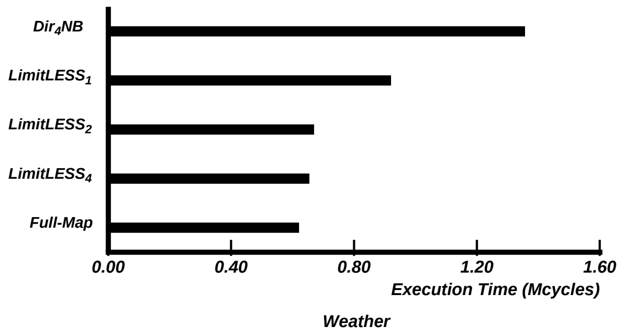

Figure 9: LimitLESS with 1, 2, and 4 hardware pointers.

LESS directories with 50 cycle emulation latency and one (LimitLESS$_1$), two (LimitLESS$_2$), and four (LimitLESS$_4$) hardware pointers. The performance of the LimitLESS protocol degrades gracefully as the number of hardware pointers is reduced. The one-pointer LimitLESS protocol is especially bad, because some of Weather's variables have a worker-set that consists of two processors.

This behavior indicates that multiprocessor software running on a system with a LimitLESS protocol will require some of the optimizations that would be needed on a system with a limited directory protocol. However, the LimitLESS protocol is much less sensitive to programs that are not perfectly optimized. Moreover, the software optimizations used with a LimitLESS protocol should not be viewed as extra overhead caused by the protocol itself. Rather, these optimizations might be employed, regardless of the cache coherence mechanism, since they tend to reduce hot-spot contention and to increase communication locality.

6 Extensions to the Scheme

Using the interface described in Section 4, the LimitLESS protocol may be extended in several ways. The simplest extension uses the LimitLESS trap handler to gather statistics about shared memory locations. For example, the handler can record the worker-set of each variable that overflows its hardware directory. This information can be fed back to the programmer or compiler to help recognize and minimize the use of such variables. For studies of data sharing, locations can be placed in the Trap-Always directory mode and handled entirely in software. This scheme permits profiling of memory transactions to these locations without degrading performance of non-profiled locations.

More interesting enhancements couple the LimitLESS protocol with the compiler and run-time systems to implement various special coherence, synchronization, and garbage-collection mechanisms. Coherence types specialized to various data types [5] can be synthesized using the Trap-Always and Trap-On-Write directory modes (defined in Section 4). For example, the LimitLESS trap handler can cause FIFO directory eviction for data structures known to migrate from processor to processor, or it can cause updates (rather than invalidates) when cached copies are modified. Synchronization objects, such FIFO lock data types, can receive special handling; the trap handler can buffer write requests for a programmer-specified variable and grant the requests on a first-come, first-serve basis.

The mechanisms implementing the LimitLESS directory protocol provide a set of generic interfaces that can be used for many different memory models. Judging by the number of synchronization and coherence mechanisms that have been defined by multiprocessor architects and programmers, it seems that there is no lack of uses for such a flexible coherence scheme.

7 Conclusion

This paper proposed a new scheme for cache coherence, called LimitLESS, which is being implemented in the Alewife machine. Hardware requirements include rapid trap handling and a flexible processor interface to the network. Preliminary simulation results indicate that the LimitLESS scheme approaches the performance of a full-map directory protocol with the memory efficiency of a limited directory protocol. Furthermore, the LimitLESS scheme provides a migration path toward a future in which cache coherence is handled entirely in software.

8 Acknowledgments

All of the members of the Alewife group at MIT helped develop and evaluate the ideas presented in this paper. Beng-Hong Lim wrote the prototype LimitLESS trap handlers. David Kranz wrote the Mul-T compiler. Beng-Hong Lim and Dan Nussbaum wrote the SPARCLE simulator, the run-time system, and the static multigrid application. Kirk Johnson supported the benchmarks. Kiyoshi Kurihara found the hot-spot variable in Weather. Gino Maa wrote the network simulator. G.N.S. Prasanna wrote the automatic matrix expression partitioner and analyzed Matexpr.

The notation for the transition state diagram borrows from the doctoral thesis of James Archibald at the University of Washington, Seattle, and from work done by Ingmar Vuong-Adlerberg at MIT.

Pat Teller and Allan Gottlieb of NYU provided the source code of the Weather and SIMPLE applications. Harold Stone and Kimming So helped us obtain the traces. The post-mortem scheduler was implemented by Mathews Cherian with Kimming So at IBM. It was extended by Kiyoshi Kurihara to include several other forms of barrier synchronization such as backoffs and software combining trees, and to incorporate feedback from the cache controller.

Machines used for simulations were donated by SUN Microsystems and Digital Equipment Corporation. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, and by grants from the Sloan Foundation and IBM.

References

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

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

[3] 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.

[4] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 355–362, New York, June 1985. IEEE.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[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] Mathews Cherian. A study of backoff barrier synchronization in shared-memory multiprocessors. Master's thesis, MIT, EECS Dept, May 1989.

[9] David R. Cheriton, Gert A. Slavenberg, and Patrick D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374, New York, June 1986. IEEE.

[10] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 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] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[13] 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.

[14] 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.

[15] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, New York, June 1985. IEEE.

[16] D. 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.

[17] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance in Large-Scale Multiprocessors. MIT VLSI Memo 1990 #90-626. Submitted for publication., October 1990.

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

[19] 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.

[20] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 348–354, New York, June 1985. IEEE.

[21] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

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

[23] 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.

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

[25] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of ASPLOS III*, pages 243–256, April 1989.