Software-Extended Coherent Shared Memory: Performance and Cost

David Chaiken and Anant Agarwal
Laboratory for Computer Science, NE43–633
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper evaluates the tradeoffs involved in the design of the software-extended memory system of Alewife, a multiprocessor architecture that implements coherent shared memory through a combination of hardware and software mechanisms. For each block of memory, Alewife implements between zero and five coherence directory pointers in hardware and allows software to handle requests when the pointers are exhausted. The software includes a flexible coherence interface that facilitates protocol software implementation. This interface is indispensable for conducting experiments and has proven important for implementing enhancements to the basic system.

Simulations of a number of applications running on a complete system (with up to 256 processors) demonstrate that the hybrid architecture with five pointers achieves between 71% and 100% of full-map directory performance at a constant cost per processing element. Our experience in designing the software protocol interfaces and experiments with a variety of system configurations lead to a detailed understanding of the interaction of the hardware and software components of the system. The results show that a small amount of shared memory hardware provides adequate performance: One-pointer systems reach between 42% and 100% of full-map performance on our parallel benchmarks. A software-only directory architecture with no hardware pointers has lower performance but minimal cost.

1 Introduction

Implementing shared memory for a large-scale multiprocessor requires balancing the performance of the system as a whole with the complexity and cost of its hardware and software components. Shared memory itself helps control the complexity of the application software written for a machine, but it requires an efficient design to achieve this goal. The Alewife architecture[3] uses a combination of hardware and software to provide shared memory at a constant cost per processing node, without sacrificing performance. Following the integrated systems approach, the architecture uses hardware to implement common memory accesses and uses software to extend the hardware by handling potentially complex scenarios.

The primary contribution of this paper is the demonstration of a complete software-extended shared memory system that allows measurement of the performance of its software components. This system proves that the software extension approach is a viable alternative for implementing a shared memory system, in terms of both cost and performance. Rather than advocating a specific machine configuration, the paper seeks to examine the performance versus cost tradeoffs inherent in implementing software-extended shared memory.

At the heart of a shared memory design lies the problem of providing fast average access time while ensuring a coherent memory model. Directory-based cache coherence protocols provide an efficient implementation of coherent shared memory for large systems. These protocols allow each processing node to take advantage of typical memory access patterns by caching frequently used data, even when the data may be shared by other nodes. A directory is a structure that helps enforce the coherence of cached data by maintaining pointers to the locations of cached copies of each memory block. When one node modifies a block of data, the memory system uses the information stored in the directory to enforce a coherent view of the data. Typically, directories are not monolithic structures but are distributed to the processing nodes along with a system's shared memory.

A promising design strategy, central to the Alewife architecture, uses a combination of hardware and software to implement a cost-efficient directory[9]. Since most data blocks in a shared memory system are shared by a small number of processing nodes[2, 32, 8], the hardware can implement a small set of pointers, and provide mechanisms to allow the system's software to extend the directory when the set of pointers is insufficient for enforcing coherence. This software-extension technique catalyzes the balance between a system's performance and cost.

LimitLESS directories, a scheme proposed in [9], is a software-extended coherence protocol that permits a tradeoff between the cost and the performance of a shared memory system. LimitLESS, which stands for a *Limit*ed directory, *L*ocally *E*xtended through *S*oftware *S*upport, implements a small number of pointers in a hardware directory (zero through five in Alewife), so that the hardware can track a few copies of any memory block. When these pointers are exhausted, the memory system hardware interrupts a local processor, thereby requesting it to maintain correct shared memory behavior by extending the hardware directory with software.

Another set of software-extended protocols (termed $\mathrm{Dir}_1\mathrm{SW}$) were proposed in [14] and [34]. These protocols use only one hardware pointer, rely on software to broadcast invalidates, and use hardware to accumulate the acknowledgments. In addition, they allow the programmer or compiler to insert Check-In/Check-Out (CICO) directives into programs to minimize the number of software traps.

All software-extended memory systems require a battery of architectural mechanisms to permit a designer to make the cost versus performance tradeoff. First, the shared memory hardware must be able to invoke extension software on the processor, and the processor must have complete access to the memory and network hardware[9, 19, 34]. Second, the hardware must guarantee forward progress in the face of protocol thrashing scenarios and high-availability interrupts[20].

Each processing node must also provide support for location-independent addressing, which is a fundamental requirement of shared memory. Hardware support for location-independent addressing permits the software to issue an address that refers to an object without knowledge of where it is resident. This hardware support includes an associative matching mechanism to detect if the object is cached, a mechanism to translate the object address and identify its home location if it is not cached, and a mechanism to issue a message to fetch the object from a remote location if it is not in local memory.

Since these mechanisms comprise the bulk of the complexity of a software-extended system, it is important to note that the benefits of these mechanisms extend far beyond the implementation of shared memory[17]. Alternative approaches to implementing shared memory proposed in [26, 30, 21] use hardware mechanisms that allocate directory pointers dynamically. These schemes do not require the mechanisms listed above, but they lack the flexibility of protocol and application software design.

A number of systems rely primarily on software to implement the mechanisms required to support shared memory [10, 23, 11, 6, 5, 4]. These systems implement coherent shared memory at low cost; however, providing location-independent addressing in software forces the granularity of data sharing to be much larger than in software-extended systems. In contrast, this paper proposes a *software-only directory architecture,* which implements location-independent addressing in hardware but relies on software to handle all inter-node memory accesses. This software-extended scheme is a low-cost alternative that allows threads to share small blocks of data.

Unlike previous studies of software-extended schemes, this paper analyzes a complete system, whose hardware is in the final stages of fabrication, and whose software is fully functional. Detailed simulations of the system lead to several conclusions about the implementation of software-extended shared memory: The minimum amount of shared memory hardware that is required to provide adequate performance is a single directory pointer (that also serves as an acknowledgment counter) per memory block. Beyond this level of hardware support, the cost and mapping of a system's DRAM become more important factors than performance. In addition, processor caches should include more associativity than a simple direct-mapped cache. Alewife uses a victim cache[16, 20] to provide the required extra associativity.

The paper extends previous work in the performance analysis of software-extended coherence protocols [9, 34] to a much wider spectrum, ranging from zero hardware pointers (the software-only directory architecture) to a full-map protocol, through the use of controlled experiments using a synthetic benchmark and a set of application programs. By studying the behavior of real software protocol handlers, this paper confirms results presented in [9] on the similarity in performance of LimitLESS_1 (one hardware pointer), LimitLESS_2, LimitLESS_4, and full-map protocols. We also confirm the findings in [34] that the performance of suitably tuned one-pointer protocols is competitive with that of multiple-pointer protocols.

In order to study the software side of hybrid shared memory systems, this paper investigates two different software systems that use the same hardware to achieve different goals. One system, written in C, incorporates a flexible coherence interface that facilitates protocol software implementation. This interface proved indispensable for conducting experiments over the whole spectrum of software-extended protocols. Our continuing research relies on the flexibility of this interface for implementing enhancements to the basic system. The other system, which is written in assembly-language, is a highly optimized and specialized implementation. While the specialized system supports only a narrow range of functionality, it shows the potential benefits of well-tuned software.

This paper presents case studies that examine how application performance varies over the spectrum of software-extended protocols. Before reaching these case studies, Section 2 gives an overview of the cost versus performance tradeoff that a software-extended memory system exploits and describes a notation for such systems. The paper's experimental methodology is then described in Section 3. Section 4 analyzes how the implementation of a software-extended protocol affects application performance; Section 5 examines the converse: how application characteristics determine protocol performance. Section 6 presents the case studies of application performance, and Section 7 suggests enhancements to improve programmability and performance. Finally, Section 8 discusses the impact of this study on the design of software-extended coherent memory systems.

2 A spectrum of protocols

The number of directory pointers that are implemented in hardware is an important design decision involved in building a software-extended shared memory system. More hardware pointers mean fewer situations in which a system must rely on software to enforce coherence, thereby increasing performance. Having fewer hardware pointers means a lower implementation cost, at the expense of reduced performance. This tradeoff suggests a whole spectrum of protocols, ranging from 0 pointers to n pointers, where n is the number of nodes in the system.

2.1 The n pointer protocol

The full-map protocol[7], which is implemented in the DASH multiprocessor[21], uses n pointers for every block of memory in the system and requires no software extension. Although this protocol permits an efficient implementation that uses only one bit for each pointer, the sheer number of pointers makes it extremely expensive for systems with large numbers of nodes. Despite the cost of the full-map protocol, it serves as a good performance goal for the software-extended schemes.

2.2 $2 \leftrightarrow (n-1)$ pointer protocols

There is a range of protocols that use a software-extended coherence scheme to implement shared memory. It is this range of protocols that allows the designer to trade hardware cost and system performance. From the point of view of implementation complexity, the protocols that implement between 2 and $n-1$ pointers in hardware are homogeneous. Of course, the $n-1$ pointer protocol would be even more expensive to implement than the full-map protocol, but it still requires exactly the same hardware and software mechanisms as the protocols at the other end of the spectrum.

The protocol extension software needs to service only two kinds of messages: read and write requests. It handles read requests by allocating an extended directory entry (if necessary), emptying all of the hardware pointers into the software structure, and recording a pointer to the node that caused the directory overflow. Subsequent requests may be handled by the hardware until the next overflow occurs. For all of these protocols, the hardware returns the appropriate data to requesting nodes; the software only needs to record requests that overflow the hardware directory.

To handle write requests after an overflow, the software transmits invalidation messages to every node with a pointer in the hardware directory or in the software directory extension. The software then returns the hardware directory to a mode that collects one acknowledgment message for each transmitted invalidation.

2.3 Zero-pointer protocols

Since the software-only directory[28] has no directory memory, it requires substantially different software than the $2 \leftrightarrow (n-1)$ range of protocols. This software must implement all coherence protocol state transitions for inter-node accesses.

While other implementations are possible, our version of the zero-pointer protocol uses one extra bit per memory block to optimize the performance of purely intra-node accesses: the bit indicates whether the associated memory block has been accessed at any time by a remote node. When the bit is clear (the default value), all memory accesses from the local processor are serviced without software traps, just as in a uniprocessor. When an inter-node request arrives, the bit is set and the extension software flushes the block from the local cache. Once the bit is set, all subsequent accesses — including intra-node requests — are handled by software extension.

2.4 One-pointer protocols

The one-pointer protocols are a hybrid of the protocols discussed above. This paper studies three variations of this class of protocols. All three use the same software routine to transmit data invalidations sequentially, but they differ in the way that they collect the messages that acknowledge receipt of the invalidations. The first variation handles the acknowledgments completely in software, requiring a trap from the hardware upon the receipt of each message. During the invalidation/acknowledgment process, the hardware pointer is unused.

The second protocol handles all but the last of a sequence of acknowledgments in hardware. If a node transmits 64 invalidations, then the hardware will process the first 63 invalidations. This variation uses the hardware pointer to store a count of the number of acknowledgments that are still outstanding. During this process, the hardware will also transmit busy messages to requesting nodes, eliminating the livelock problem. Upon receiving the 64^{th} acknowledgment, the hardware invokes the software, which takes care of transmitting data to the requesting node.

The third protocol handles all acknowledgment messages in hardware. This protocol actually requires storage for two hardware pointers: one pointer to store the requesting node's identifier and another to count acknowledgments. Although a designer would always choose to implement a two-pointer protocol over this variation of the one-pointer protocol, it still provides a useful baseline for measuring the performance of the other two variations.

2.5 A notation for the spectrum

We now introduce a notation that allows us to articulate clearly the differences between various implementations and facilitates a precise cost comparison.

Our notation is derived from a nomenclature for directory-based coherence protocols introduced in [2]. In the previous notation, a protocol was represented as $\mathrm{Dir}_i\mathrm{X}$, where i represented the number of explicit copies tracked, and X was B or *NB* depending on whether or not the protocol issued broadcasts. Notice that this nomenclature does not distinguish between the functionality implemented in the software and in the hardware. Our notation attempts to capture the spectrum of features of software-extended protocols that have evolved over the past several years, and previously termed $\mathrm{LimitLESS}_1$, $\mathrm{LimitLESS}_4$, and others in [9], and $\mathrm{Dir}_1\mathrm{SW}$, $\mathrm{Dir}_1\mathrm{SW}+$, and others in [14, 34].

For both hardware and software, our notation divides the mechanisms into two classes: those that dictate directory actions upon receipt of processor requests, and those that dictate directory actions for acknowledgments.

Accordingly, our notation specifies a protocol as: $\mathbf{Dir_i H_X S_{Y,A}}$, where i is the number of explicit pointers recorded by the system – in hardware or in software – for a given block of data.

The parameter X is the number of pointers recorded in a hardware directory when a software extension exists. X is *NB* if the number of hardware pointers is i and no more than i shared copies are allowed, and is B if the number of hardware pointers is i and broadcasts are used when more than i shared copies exist. Thus the full-map protocol in DASH [21] is termed $Dir_n H_{NB} S_{-}$.

The parameter Y is *NB* if the hardware-software combination records i explicit pointers and allows no more than i copies. Y is B if the software resorts to a broadcast when more than i copies exist.

The A parameter is *ACK* if a software trap is invoked on *every* acknowledgment. A missing A field implies that the hardware keeps an updated count of acknowledgments received. Finally, the A parameter is *LACK* if a software trap is invoked only on the *last* acknowledgment.

According to this notation, the $\mathrm{LimitLESS}_1$ protocol defined in [9] is termed $Dir_n H_1 S_{NB}$, denoting that it records n pointers, of which only one is in hardware. The hardware handles all acknowledgments and the software issues invalidations to shared copies when a write request occurs after an overflow. This paper deals with three variants of the one-pointer protocols defined above. In our notation, the three one-pointer protocols are $Dir_n H_1 S_{NB,ACK}$, $Dir_n H_1 S_{NB,LACK}$, and $Dir_n H_1 S_{NB}$, respectively.

The set of software-extended protocols introduced in [14] and [34] can also be expressed in terms of our notation. The $\mathrm{Dir}_1\mathrm{SW}$ protocol maintains one pointer in hardware, resorts to software broadcasts when more than one copy exists, and counts acknowledgments in hardware. In addition, their protocol traps into software on the last acknowledgment[33]. In our notation, this protocol is represented as $Dir_1H_1S_{B,LACK}$. This protocol is different from the $Dir_nH_1S_{NB,LACK}$ protocol in that $Dir_1H_1S_{B,LACK}$ maintains only one explicit pointer, while $Dir_nH_1S_{NB,LACK}$ maintains one pointer in hardware and extends the directory to n pointers in software. An important consequence of this difference is that the $Dir_nH_1S_{NB,LACK}$ potentially traps on read requests, while $Dir_1H_1S_{B,LACK}$ does not. Unlike $Dir_nH_1S_{NB,LACK}$, $Dir_1H_1S_{B,LACK}$ must issue broadcasts on write requests to memory blocks that are cached by multiple nodes.

3 Methodology

This section first describes the MIT Alewife machine, which provides a proof of concept for software-extended memory systems and a platform for experimenting with many aspects of multiprocessor design and programming. While the machine supports an interesting range of protocols, it does not implement the full spectrum of software-extended schemes that this paper evaluates. Only a simulation system can provide the range of protocols, the deterministic behavior, and the non-intrusive observation functions that are required for analyzing the spectrum of software-extended protocols. The second half of this section describes the simulation system used to do the experiments discussed in the remainder of the paper.

3.1 The Alewife Machine

Alewife is a large-scale multiprocessor with distributed shared memory. Figure 1 shows an enlarged view of a node in the Alewife machine. Each node consists of a 33 MHz Sparcle processor[1], 64K bytes of direct-mapped cache, 4 Mbytes of globally-shared main memory, and a floating-point coprocessor. The nodes communicate via messages through a network[29] with a mesh topology. A single-chip communications and memory management (CMMU) on each node holds the cache tags and implements the memory coherence protocol by synthesizing messages to other nodes. All of the node components, with the exception of the CMMU, have been fabricated and tested. The CMMU is in fabrication.

In order to provide a platform for shared memory research, Alewife supports dynamic reconfiguration of coherence protocols on a block-by-block basis. The machine supports $Dir_nH_0S_{NB,ACK}$, $Dir_nH_2S_{NB}$, $Dir_nH_3S_{NB}$, $Dir_nH_4S_{NB}$, $Dir_nH_5S_{NB}$, $Dir_5H_5S_B$ and a variety of other protocols. The node diagram in Figure 1 illustrates a memory block with two hardware pointers and an associated software-extended directory structure ($Dir_nH_2S_{NB}$). The current default boot sequence configures every block of shared memory with a $Dir_nH_5S_{NB}$ protocol, which uses all of the available hardware pointers.

In addition to the standard hardware pointers, Alewife implements a special one-bit pointer for the node that is local to the directory. Several simulations show that this extra pointer improves performance by only about 2%. Its main benefit lies in reducing the complexity of the protocol hardware and software by

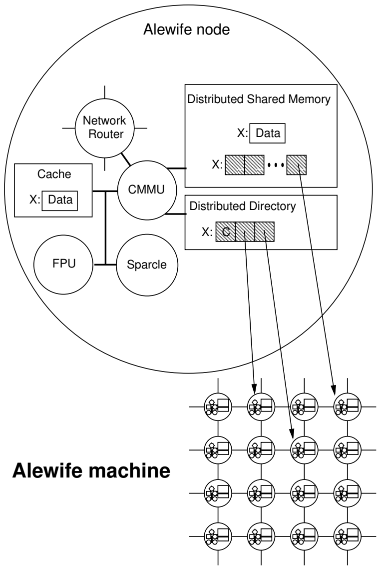

Figure 1: Alewife node, with a $Dir_nH_2S_{NB}$ memory block.

eliminating the possibility that a node will cause its local hardware directory to overflow. For the results presented in this paper, all of the protocols (except $Dir_nH_0S_{NB,ACK}$) use the one-bit pointer in addition to the normal hardware pointers.

3.2 NWO: the Alewife simulator

NWO is a multi-purpose simulator that provides a deterministic debugging and test environment for the Alewife machine. The simulator performs a cycle-by-cycle simulation of all of the components in Alewife. NWO is binary compatible with Alewife's hardware: programs that run on the simulator will be able to run on the actual machine *without recompilation.* The CMMU protocol state-transition tables are automatically compiled from the hardware specification into a simulator executable format, so that NWO incorporates the hardware protocol directly. NWO models the Alewife data paths accurately enough that it is used to drive the transistor-level simulations of the CMMU. Although Alewife does not support one-pointer protocols or protocols with more than five hardware pointers, NWO has been extended to support a complete spectrum of software-extended protocols, from $Dir_nH_0S_{NB,ACK}$ to $Dir_nH_{NB}S_{-}$.

There are two inaccuracies in the simulation. First, NWO does not model the Sparcle or FPU pipelines, even though it does model many of the pipelined data paths within the CMMU. Second, NWO models communication contention at the CMMU network transmit and receive queues, but does not model contention within the network switches.

The initial implementation of NWO targeted SPARC and MIPS-based workstations; we have also developed a version of the simulator that runs on Thinking Machines' CM-5 multiprocessors. In the latter implementation, each CM-5 node simulates the processor, memory, and network hardware of one or more Alewife nodes. The CM-5 port of our simulator has proved invaluable, especially for running simulations of 64 and 256 node Alewife systems.

4 Software interfaces

Two different versions of the protocol extension software have been written for the Alewife machine. One version incorporates a flexible coherence interface that allows rapid protocol implementation in the C programming language. The other version is a set of protocol handlers that are implemented in assembly language and are carefully tuned to take full advantage of the features of the Alewife architecture.

4.1 Protocol software implementations

The C version of the Alewife protocol extension software implements the whole range of software-extended protocols within a flexible framework. A single set of C routines implements all of the protocols from $Dir_nH_2S_{NB}$ to $Dir_nH_{NB}S_{-}$. Other modules linked into the same kernel support $Dir_nH_0S_{NB,ACK}$, $Dir_nH_1S_{NB}$, $Dir_nH_1S_{NB,LACK}$, and $Dir_nH_1S_{NB,ACK}$.

The flexible interface facilitates the construction of all of these protocols by providing C macros for hardware directory manipulation, protocol message transmission, a free-listing memory manager, and hash table administration. The interface eliminates the need for the protocol designer to understand many of the details of the Alewife hardware implementation. For example, the protocol interface sets up an environment that lets the protocol designer treat every protocol event as if it were generated by an asynchronous inter-node request.

In addition, the framework hides other implementation details such as atomic protocol transitions and livelock situations. The framework ensures the atomicity of protocol transitions by guaranteeing that asynchronous messages in a CMMU internal queue are processed before handling synchronous events. Livelock situations can occur when protocol software-extension requests occur so frequently that user code cannot make forward progress. The framework solves this problem by using a timer interrupt to implement a watchdog that detects possible livelock, temporarily shuts off asynchronous events, and allows the user code to run unmolested. In practice, such conditions happen only for $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_1S_{NB,ACK}$, when they handle acknowledgments in software. The framework provides a very simple interface that allows these protocols to invoke the watchdog directly.

During the course of the study reported in this paper, the flexible coherence interface proved itself to be an indispensable tool for rapidly prototyping a complete set of protocols. The framework is currently being used to implement some of the enhancements to the basic protocols that are described in Section 7.

Unfortunately, flexibility comes at the cost of performance. All of the mechanisms that protect the protocol designer from the details of the Alewife hardware implementation increase the time that it takes to handle protocol requests in software. An assembly-language version of the Alewife protocol extension handlers helps investigate the performance versus flexibility tradeoff by optimizing the performance of the software. Since this approach requires a large programming effort, this version only implements $Dir_nH_5S_{NB}$.

The code for this optimized version is hand-tuned to keep instruction counts to a minimum. To reduce memory management time, it uses a special free-list of extended directory structures that are initialized when the kernel boots the machine. The assembly-language version also takes advantage of a feature of Alewife's directory that eliminates the need for a hash table lookup.

4.2 Comparing the implementations

The primary difference between the performance of the two implementations of the protocol extension software is the amount of time that it takes to process a protocol request. Table 1 gives the average number of cycles required to process $Dir_nH_5S_{NB}$ read and write requests for both of the implementations. These software handling latencies were measured by running the WORKER benchmark (described in the next section) on a 16 node system. The latencies are relatively independent of the number of nodes that read each memory block. In most cases, the hand-tuned version of the software reduces the latency of protocol request handlers by about a factor of two.

Readers Per Block	C Read Request	Assembly Read Request	C Write Request	Assembly Write Request
8	436	162	726	375
12	397	141	714	393
16	386	138	797	420

Table 1: Average software extension latencies for C and for assembly language, in execution cycles.

These latencies may be understood better by analyzing the number of cycles spent on each activity required to extend a protocol in software. Table 2 accounts for all of the cycles spent in a read and a write request from both versions of the protocol software. These counts come from cycle-by-cycle traces of read and write requests with eight readers and one writer per memory block. In order to select a representative individual from each sample, we choose a median request of each type (as opposed to the average, which we use above to summarize aggregate behavior).

The dispatch and trap return activities are standard sequences of code that invoke hardware exception and interrupt handlers and allow them to return to user code, respectively. (The dispatch activity does not include the three cycles that Sparcle takes to flush its pipeline and to load the first trap instruction.) In the assembly-language version, these sequences are streamlined to invoke the protocol software as quickly as possible. The C implementation of the software requires an extra protocol-specific dispatch in order to set up the C environment and hide the details of the Alewife hardware. For the types of protocol requests that occur when running the WORKER benchmark, this extra overhead does not significantly impact performance. The extra code in the C version that supports the non-Alewife protocols implemented only in the simulator also impacted performance minimally.

The difference between the performance of the C and assembly-language protocol handlers lies in the flexibility of the C interface. The assembly-language version avoided most of the expense of memory management and hash table administration by implementing a special-purpose solution to the directory structure allocation and lookup problem. This solution relies heavily on the format

Activity	C Read Request	Assembly Read Request	C Write Request	Assembly Write Request
trap dispatch	11	11	9	11
system message dispatch	14	15	14	15
protocol-specific dispatch	10	N/A	10	N/A
decode and modify hardware directory	22	17	52	40
save state for function calls	24	N/A	17	N/A
memory management	60	65	28	11
hash table administration	80	N/A	74	N/A
store pointers into extended directory	235	74	99	45
invalidation lookup and transmit	N/A	N/A	419	251
support for non-Alewife protocols	10	N/A	6	N/A
trap return	14	11	9	11
total (median latency)	480	193	737	384

Table 2: Breakdown of execution cycles measured from median-latency read and write requests. Each memory block has 8 readers and 1 writer. N/A stands for not applicable.

of Alewife's coherence directory and is not robust in the context of a system that runs a large number of different applications over a long period of time. However, it does place a minimum bound on the time required to perform these tasks. As the Alewife system evolves, critical pieces of the protocol extension software that are implemented under the flexible interface will be hand-tuned to realize the best of both worlds.

5 Worker sets and performance

A *worker set* is defined to be the set of nodes that simultaneously access a unit of data. The software-extension approach is predicated on the observation that, for a large class of applications, most worker sets are relatively small. Small worker sets are handled in hardware by a limited directory structure. Memory blocks with large worker sets must be handled in software, at the expense of longer memory access latency and processor cycles that are spent on protocol handlers rather than on user code.

This section uses a synthetic benchmark to investigate the relationship between an application's worker sets and the performance of software-extended coherence protocols. The benchmark, called WORKER, uses a data structure that creates memory blocks with an exact worker set size. WORKER consists of an initialization phase that builds the worker set data structure and a number of iterations that perform repeated memory accesses to the structure.

The nodes begin each iteration by reading the appropriate slots in the worker set structure. After the reads, they execute a barrier and then perform the writes to the structure. Finally, the nodes execute a barrier and continue with the next iteration. Every read request causes a cache miss and every write request causes a directory protocol to send exactly one invalidation message to each reader. This completely deterministic memory access pattern provides a controlled experiment for comparing the performance of different protocols.

In order to analyze the relationship between worker set sizes and the performance of software-extended shared memory, we perform

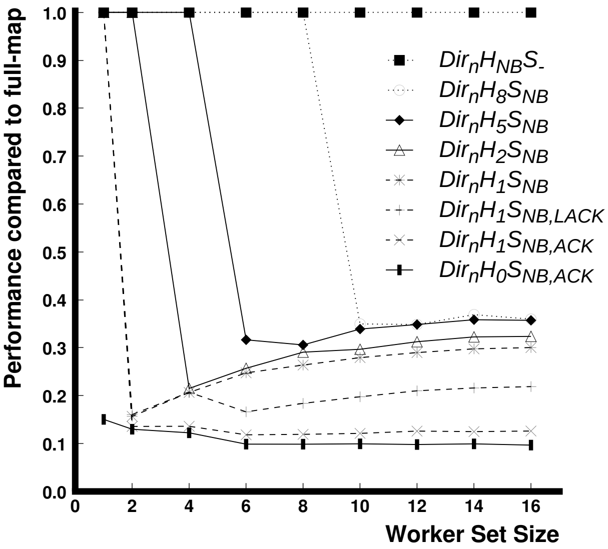

Figure 2: Protocol performance and worker set size.

simulations of WORKER running on a range of protocols. The simulations are restricted to a relatively small system because the benchmark is both regular and completely distributed, so the results would not be qualitatively different for a larger number of nodes.

Figure 2 presents the results of a series of 16 node simulations. The horizontal axis gives the size of the worker sets generated by the benchmark. The vertical axis measures the ratio of the run-time of each protocol and the run-time of a full-map protocol ($Dir_nH_{NB}S_-$) running the same benchmark configuration.

The solid curves in Figure 2 indicate the performance of some of the protocols that are implemented in the actual Alewife machine. As expected, the more hardware pointers, the better the performance of the software-extended system. The performance of $Dir_nH_5S_{NB}$ is particularly easy to interpret: its performance is exactly the same as the full-map protocol up to a worker set size of 4, because the worker sets fit entirely within the hardware directory. For small worker set sizes, software is never invoked. The performance of $Dir_nH_5S_{NB}$ drops as the worker set size grows, due to the expense of handling memory requests in software.

At the other end of the performance scale, the $Dir_nH_0S_{NB,ACK}$ protocol performs significantly worse than the other protocols, for all worker set sizes. Since WORKER is a shared memory stress test and exaggerates the differences between the protocols, Figure 2 shows the worst possible performance of the software-only directory. The measurements in the next section, which experiment with more realistic applications, yield a more optimistic outlook for the zero and one-pointer protocols.

The dashed curves correspond to one-pointer protocols that run only in the simulation environment. These three protocols differ only in the way that they handle acknowledgment messages (see Section 2.4). For all non-trivial worker set sizes, the protocol that traps on every acknowledgment message ($Dir_nH_1S_{NB,ACK}$) performs significantly worse than the protocols that can count acknowledgments in hardware. $Dir_nH_1S_{NB}$, which never traps on acknowledgment messages, has very similar performance to the $Dir_nH_2S_{NB}$ protocol, except when running with size 1 worker sets. Since this version of $Dir_nH_1S_{NB}$ requires the same amount of directory storage as $Dir_nH_2S_{NB}$, the similarity in performance is not surprising.

Of the three different one-pointer protocols, the protocol that traps only on the last acknowledgment message in a sequence ($Dir_nH_1S_{NB,LACK}$) makes the most cost-efficient use of the hardware pointers. This efficiency comes at a slight performance cost. For the WORKER benchmark, this protocol performs between 0% and 50% worse than $Dir_nH_1S_{NB}$. When the worker set size is 4 nodes, $Dir_nH_1S_{NB,LACK}$ actually performs slightly better than $Dir_nH_1S_{NB}$. This anomaly is due to a memory-usage optimization that attempts to reduce the size of the software-extended directory when handling small worker sets. The optimization, implemented in the $Dir_nH_1S_{NB,LACK}$, $Dir_nH_1S_{NB,ACK}$ and $Dir_nH_0S_{NB,ACK}$ protocols, improves the run-time performance of all three protocols for worker set sizes of 4 or less.

6 Application case studies

This section presents more practical case-studies of several programs and investigates how the performance of applications depends on memory access patterns, the coherence protocol, and other machine parameters.

The names and characteristics of the applications we analyze are given by Table 3. They are written in C, Mul-T[18] (a parallel dialect of LISP), and Semi-C[15] (a language akin to C with support for fine-grain parallelism). Each application (except MP3D) is studied with a problem size that realizes more than 50% processor utilization on a simulated 64 node machine with a full-map directory. All performance results use protocols implemented with the flexible coherence interface described in Section 4.

Figure 4 presents the basic performance data for the applications running on 64 nodes. The horizontal axis shows the number of directory pointers implemented in hardware, thereby measuring the cost of the system. The vertical axis shows the speedup of the mul-

Name	Language	Size	Sequential
TSP	Mul-T	10 city tour	1.1 sec
AQ	Semi-C	*see text*	0.9 sec
SMGRID	Mul-T	129×129	3.0 sec
EVOLVE	Mul-T	12 dimensions	1.3 sec
MP3D	C	10,000 particles	0.6 sec
WATER	C	64 molecules	2.6 sec

Table 3: Characteristics of applications. Sequential time assumes a clock speed of 33MHz.

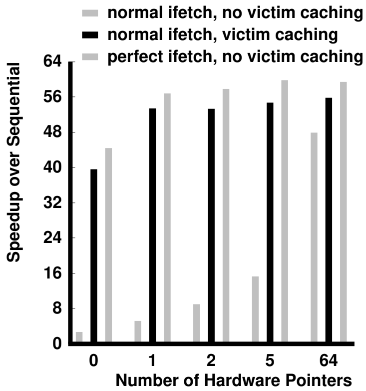

Figure 3: TSP: detailed 64 node performance analysis.

tiprocessor execution over a sequential run without multiprocessor overhead. The software-only directory is always on the left and the full-map directory on the right. All of the figures in this section show $Dir_nH_1S_{NB,ACK}$ performance for the one-pointer protocol.

The most important observation is that the performance of $Dir_nH_5S_{NB}$ is always between 71% and 100% of the performance of $Dir_nH_{NB}S_{-}$. Thus, the data in Figure 4 provides strong evidence that the software extension approach is a viable alternative for implementing a shared memory system. The rest of this section seeks to provide a more detailed understanding of the performance of software-extended systems.

Traveling Salesman Problem TSP solves the traveling salesman problem using a branch-and-bound graph search. The application is written in Mul-T and uses the `future` construct to specify parallelism. In order to ensure that the amount of work performed by the application is deterministic, we seed the best path value with the optimal path. Given the characteristics of the application's memory access pattern, one would expect TSP to perform well with a software-extended protocol: the application has very few large worker sets. In fact, most – but not all – of the worker sets are small sets of nodes that concurrently access partial tours.

Figure 3 presents detailed performance data for TSP running on a 64 node machine. Contrary to our expectations, TSP suffers severe performance degradation when running with the software-extended protocols. The gray bars in the figure show that the five-pointer protocol performs more that 3 times worse than the full-map protocol. This performance decrease is due to instruction/data thrashing in Alewife's combined, direct-map caches: When we profiled the address reference pattern of the application, we found

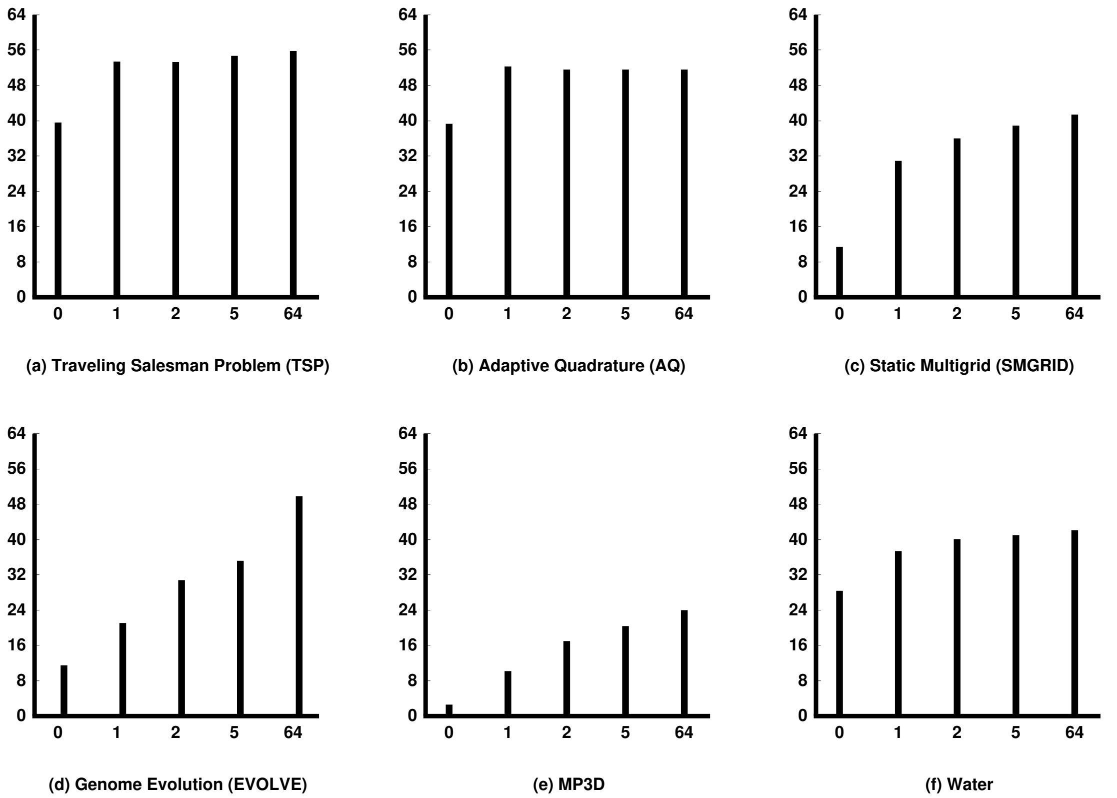

Figure 4: Application speedups over sequential, running on 64 nodes. Horizontal axis shows number of hardware pointers. Vertical axis shows speedup over sequential execution.

that two memory blocks that were shared by *every* node in the system were constantly replaced in the cache by commonly run instructions.

In order to confirm this observation, we invoked a simulator option that allows one-cycle access to every instruction without using the cache. This option, called *perfect ifetch*, eliminates the effects of instructions on the memory system. The hashed bars in Figure 3 confirm that instruction/data thrashing was a serious problem in the initial runs. Absent the effects of instructions, all of the protocols except the software-only directory realize performance equivalent (within experimental error) to a full-map protocol.

While perfect instruction fetching is not possible in real systems, there are various methods for relieving instruction/data thrashing by increasing the associativity of the cache system. Alewife's approach to the problem is to implement a version of victim caching[16], which uses the transaction store[20] to provide a small number of buffers for storing blocks that are evicted from the cache. The black bars in Figures 4(a) and 3 show the performance for TSP on a system with victim caching enabled. The few extra buffers improve the performance of the full-map protocol by 16%, and allow all of the protocols with hardware pointers to perform about as well as full-map. For this reason, the studies of all of the other applications in this section enable victim-caching by default.

It is interesting to note that $Dir_nH_0S_{NB,ACK}$ with victim caching achieves almost 70% of the performance of $Dir_nH_{NB}S_-$. This low-cost alternative seems viable for applications with limited amounts of sharing.

Thus far, we have compared the performance of the protocols under an environment where the full-map protocol achieves close to maximum speedup. On an application that requires only 1 second to run, the system with victim caching achieves a speedup of about 55 for the 5 pointer protocol. In order to investigate the effects of running an application with suboptimal speedups, we ran the same problem size on a 256 node machine with victim caching enabled. Figure 5 shows the results, which indicate a speedup of 142 for full-map and 134 for five-pointers. We consider these speedups remarkable for this problem size and note that the software-extended system performs only 6% worse than full-map in this configuration. The difference in performance is due primarily to the increased contribution of the transient effects over distributing data to 256 nodes at the beginning of the run.

Adaptive Quadrature AQ performs numerical integration of bivariate functions using adaptive quadrature. The core of the algorithm is a function that integrates the range under a curve by recursively calling itself to integrate sub-ranges of that range. The function used for this study is x^4y^4, which is integrated over the square $((0,0),(2,2))$ with an error tolerance of 0.005.

Since all of the communication in the application is producer-consumer, we expect this application to perform equally well for all protocols that implement at least one directory pointer in hardware. Figure 4(b) confirms this expectation by showing the performance of the application running on 64 nodes. Again, $Dir_nH_0S_{NB,ACK}$ performs respectably due to the favorable memory access patterns

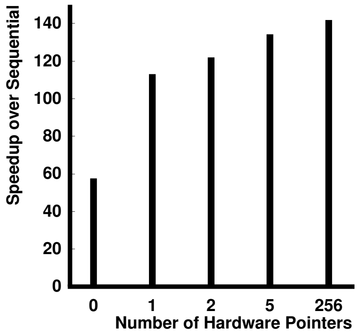

Figure 5: TSP running on 256 nodes.

in the application.

Static Multigrid SMGRID uses the multigrid method to solve elliptical partial differential equations[13]. The algorithm consists of performing a series of Jacobi-style iterations on multiple grids of varying granularities. The speedup over sequential is limited by the fact that only a subset of nodes work during the relaxation on the upper levels of the pyramid of grids. Furthermore, data is more widely shared in this application than in either TSP or AQ. The consequences of these two factors appear in Figure 4(c): the absolute speedups are lower than either of the previous applications, even though the sequential time is three times longer.

The larger worker set sizes of multigrid cause the performance of the different protocols to separate. $Dir_nH_0S_{NB,ACK}$ performs more than three times worse than the full-map protocol. The others range from 25% worse in the case of $Dir_nH_1S_{NB,ACK}$ to 6% worse in the case of $Dir_nH_5S_{NB}$.

Genome Evolution EVOLVE is a graph traversal algorithm for simulating the evolution of genomes, which is reduced to the problem of traversing a hypercube and finding local and global maxima. The application searches for a path from the initial conditions to a local fitness maximum.

Of all of the applications in Figure 4, EVOLVE causes $Dir_nH_5S_{NB}$ to exhibit the worst performance degradation compared to $Dir_nH_{NB}S_-$: the worker sets of EVOLVE seriously challenge a software-extended system. Figure 6 shows the number of worker sets of each size at the end of a 64 node run. Note that the vertical axis is logarithmically scaled: there are almost 10,000 one-node worker sets, while there are 25 worker sets of size 64. The significant number of nontrivial worker sets implies that there should be a sharp difference between protocols with different numbers of pointers. The large worker sets sizes impact the 0 and 1 pointer protocols most severely. Thus, EVOLVE provides a good example of a program that can benefit from a system's hardware directory pointers.

MP3D The MP3D application is part of the SPLASH parallel benchmark suite [31]. For our simulations, we use a problem size of 10,000 particles, turn the locking option off, and augment the standard `p4` macros with Alewife's parallel C library[25]. Since this application is notorious for exhibiting low speedups [22], the results in Figure 4(e) are encouraging: $Dir_nH_{NB}S_-$ achieves a speedup of 24 and $Dir_nH_5S_{NB}$ realizes a speedup of 20. These speedups are for a relatively small problem size, and we expect absolute speedups to increase with problem size.

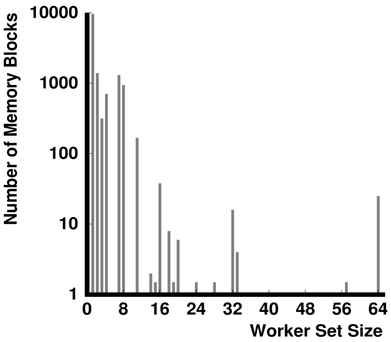

Figure 6: Histogram of worker set sizes for EVOLVE, running on 64 nodes.

The software-only directory exhibits the worst performance (only 11% of the speedup of full-map) on MP3D. Thus, MP3D provides another example of an application that can benefit from at least a small number of hardware directory pointers.

Water The Water application, also from the SPLASH application suite, is run with 64 molecules. In addition to the `p4` macros, this version of Water uses Alewife's parallel C library for barriers and reductions. Figure 4(f) shows that all of the software-extended protocols provide good speedups for this tiny problem size. Once again, the software-only directory offers almost 70% of the performance of the full-map directory.

7 Enhancement opportunities

This paper uses a basic definition of software-extended coherent shared memory in order to analyze the viability of the approach. The $Dir_iH_XS_{Y,A}$ notation itself implies a straightforward software directory extension. While this definition allows for a relatively simple analysis technique, the true power of the software-extension approach lies in deviating from the basic implementation. The generality of the flexible coherence interface described in Section 4 provides a platform for experimenting with schemes that enhance the performance and the functionality of the base protocols.

[9] suggests several extensions to the basic software such as a FIFO lock data type. To date, the protocol extension software has been used to implement a FIFO lock data type, stack overflow exceptions, and a fast barrier implementation. These enhancements are aimed at providing efficient functions that improve the programmability of the machine. [9] also indicates that LimitLESS software could be enhanced to improve the performance of normal shared memory variables, such as variables with large worker sets. The following types of extensions give examples of current research:

Program and compiler annotations Program annotations allow a programmer to give the system information about the way that an application interacts with shared memory. [14] and [34] propose and evaluate this method for improving the performance of software-extended shared memory. The studies show that given appropriate annotations, a large class of applications can perform well on $Dir_1H_1S_{B,LACK}$. [24] demonstrates a compiler annotation scheme for optimizing the performance of protocols that dynamically allocate directory pointers.

Dynamic detection [12] and [27] propose a hardware mechanism that dynamically adapts to migratory data. Protocol extension software could perform similar optimizations. In addition, there are some classes of data that create severe performance bottlenecks. These classes tend to be the result of a simplistic programming style or a performance bug. Examples of these widely-shared data structures include synchronization objects, work queues, and frequently-written global objects. Preliminary results from our experiments show that protocol extension software may improve performance for this type of data by dynamically selecting sequential or parallel invalidation procedures.

Profile, detect, and optimize Some types of data do not create serious performance bottlenecks, but can benefit from optimization. An example of this class of data is widely-shared, read-only data. During the development phase of an application, enhanced protocol software could be used in a profiling mode to detect the existence of read-only data. The system could use the information to optimize the production version of the application.

Data specific Some types of data might be hard to optimize automatically, either dynamically or statically. In this case, a user could select special coherence types from a library, or even write an application-specific protocol under the flexible coherence interface.

8 Conclusions

The software extension approach offers a cost-efficient method for implementing scalable, coherent, high-performance shared-memory. Experience with the design of such a system shows that a minimum of one directory pointer and an acknowledgment counter should be implemented in hardware. Since all of the protocols that implement small numbers of hardware directory pointers have similar performance, factors such as the cost and mapping of each node's DRAM will dominate performance considerations when building a software-extended system.

The hardware components of a software-extended system must be tuned carefully to achieve high performance. Since the software-extended approach increases the penalty of cache misses, thrashing situations cause particular concern. Adding extra associativity to the processor side of the memory system, by implementing victim caches or by building set-associative caches, can dramatically decrease the effects of thrashing on the system as a whole.

Experiments with the implementation of protocol software indicate that such systems should include a flexible coherence interface that facilitates the implementation of specialized protocols. Such protocols could enhance the basic protocol software to improve both the programmability of machines and the performance of shared memory.

Acknowledgments

The research reported in this paper would not have happened without the support of the members of the Alewife group at MIT. John Kubiatowicz designed and implemented Alewife's CMMU, which is the heart of the LimitLESS system. David Kranz and Beng-Hong Lim wrote much of Alewife's run-time system, parts of NWO, and helped debug the protocol extension software. John Piscitello wrote an early version of the $Dir_N H_0 S_{NB,ACK}$ protocol. Beng-Hong Lim's efforts made the benchmarks (written by Kirk Johnson, Dan Nussbaum, and Anshu Aggarwal) available for this paper.

Kirk Johnson wrote and stabilized NWOP, the CM-5 version of NWO. We would like to thank Alan Mainwaring, Dave Douglas, and Thinking Machines Corporation for their generosity and assistance in porting our simulation system to the CM-5. Additional thanks to Thinking Machines Corporation (especially the folks who maintain the in-house machines) for allowing us to use many late-night CM-5 cycles during the results-generation phase of this research.

Dana Henry, Kirk Johnson, Steve Keckler, Michael Noakes, and the anonymous referees gave helpful comments during the production of this paper.

This research has been supported by NSF grant # MIP-9012773, ARPA grant #N00014-91-J-1698, and a NSF Presidential Young Investigator Award.

References

[1] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[2] 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*, pages 280–289, New York, June 1988. IEEE.

[3] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[4] Henri E. Bal and M. Frans Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93).*, September 1993.

[5] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON'93)*, pages 528–537. IEEE, February 1993.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of MUNIN. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] A. Cox and R. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989. Also as a Univ. Rochester TR-263, May 1989.

[12] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherence for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual Symposium on Computer Architecture 1993*, New York, May 1993. ACM.

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

[14] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 262–273, Boston, October 1992. ACM.

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

[16] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings, International Symposium on Computer Architecture '90*, pages 364–373, June 1990.

[17] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Practice and Principles of Parallel Programming (PPoPP) 1993*, pages 54–63, San Diego, CA, May 1993. ACM. Also as MIT/LCS TM-478, January 1993.

[18] 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.

[19] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference (ISC) 1993*, Tokyo, Japan, July 1993. IEEE. Also as MIT/LCS TM, December 1992.

[20] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, Boston, October 1992. ACM.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[22] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–60, January 1993.

[23] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *International Conference on Parallel Computing*, pages 94–101, 1988.

[24] David J. Lilja and Pen-Chung Yew. Improving Memory Utilization in Cache Coherence Directories. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1130–1146, October 1993.

[25] Beng-Hong Lim. Functions for Parallel C on the Alewife System. ALEWIFE Memo No. 37, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1993.

[26] 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*, pages 138–147, New York, June 1990. IEEE.

[27] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual Symposium on Computer Architecture 1993*, New York, May 1993. ACM.

[28] John D. Piscitello. A Software Cache Coherence Protocol for Alewife. Master's thesis, MIT, Department of Electrical Engineering and Computer Science, May 1993.

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

[30] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proceedings International Symposium on Shared Memory Multiprocessing*, Japan, April 1991. IPS Press.

[31] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[32] 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.

[33] David A. Wood. Private Communication, October 1993.

[34] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *In Proceedings of the 20th Annual International Symposium on Computer Architecture 1993*, pages 156–167, San Diego, CA, May 1993. ACM.