Architectural Support for an Efficient Implementation of a Software-Only Directory Cache Coherence Protocol

Håkan Grahn and Per Stenström

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden

June, 1995

Abstract

S*oftware-only directory cache coherence protocols* emulate directory management by handlers executed on the compute processor in shared-memory multiprocessors. While their potential lies in lower implementation cost and complexity than traditional hardware-only directory protocols, the miss penalty for cache misses induced by application data accesses as well as directory accesses is a critical issue to address.

In this paper, we study important support mechanisms for software-only directory protocols in the context of a processor node organization for a cache-coherent NUMA architecture. We find that it is possible to remove or hide software handler latency for local as well as remote read misses by adopting simple hardware support mechanisms. To further reduce the overhead of software handler execution, we study the effects of directory data caching. While this could pollute the caches, our results suggest that this effect is marginal and that software handler execution overhead is drastically reduced by allowing caching. Overall, the software-only directory protocol enhanced with the miss handling support mechanisms exhibits a performance that is competitive with hardware-only protocols but with a lower cost and complexity.

1 Introduction

While private caches and a directory-based coherence protocol constitute a key approach to build efficient shared-memory multiprocessors, cost, complexity, and inflexibility of “hardware-only” directory mechanisms as in Stanford DASH [16] make them less attractive. *Software-only directory protocols* [5] reduce the complexity and promote flexibility by migrating directory management to software handler execution on the compute processors to the expense of lower performance.

This paper explores three major sources of lower performance for a software-only directory protocol. First, in a previous work [7], we have found that an important performance limitation of software-only directory protocols is caused by the software-handler latency that ends up on the memory access path of *remote misses*, i.e., misses that are serviced by another processor node. We found that this latency component can be hidden by employing parallelism between inter-node data transfers and directory operations; a strategy that requires some hardware modifications that we did not explore in depth in [7]. Second, another performance issue left unexplored in [7] is how to avoid that the software-handler latency ends up on the memory-access path of *local misses* as well. Third, a significant part of the software handler latency is associated with accesses to the directory which potentially could be done more efficiently if directory entries are cached. However, cached directory entries may contend with application data making the overall running time longer so an important question becomes whether directory entries should be cached or not.

In this paper, we study efficient support for miss handling in the context of a detailed architecture of a shared-memory processor node that implements a software-only directory protocol. In this process, we also pinpoint architectural issues such as deadlock/livelock situations and find that while some of them are unique for software-only directory protocols, there are known solutions to circumvent them. Moreover, by comparing the hardware mechanisms tailored for efficient miss handling in the software-only design with those responsible for directory management in the hardware-only design, we argue that the processor node for software-only directory protocols has lower cost and complexity.

As for the performance side of the argument, we simulate processor nodes that implement hardware-only as well as software-only directory protocols using four scientific benchmark programs from the SPLASH suite [20]. We find that the software-only directory protocol using the mechanisms for efficient miss handling of local and remote application data misses is competitive with the hardware-only protocol. Moreover, our data suggest that directory caching can improve overall application performance significantly; it only marginally interferes with application data while significantly reducing the software handler latency.

The rest of the paper is organized as follows. In the next section we present our architectural framework and how the baseline hardware-only and software-only directory coherence protocol are implemented within this framework. Then, in Section 3 we explain why application data misses as well as directory data accesses can limit the performance of software-only directory protocols. We also identify the hardware mechanisms needed to make them more efficient. Section 4 establishes our experimental set-up, whereas Section 5 specifically addresses the performance issues raised in this study. Finally, many research efforts aim at studying the hardware/software cache protocol trade-off and we discuss them in the context of our work in Section 6 before reaching our conclusions in Section 7.

2 Architectural Framework and Coherence Protocol

Our architectural framework is a cache-coherent non-uniform memory access (CC-NUMA) architecture with a number of processor nodes connected by a mesh network and cache coherence is maintained by a write-invalidate protocol. We start in Section 2.1 by describing the architectural framework which is the same for the baseline hardware-only and software-only designs we simulate. Section 2.2 then describes the mechanisms needed by the baseline hardware-only directory protocol whereas Section 2.3 focuses on the architectural differences to support the baseline software-only directory protocol we simulate.

2.1 An Architectural Framework for HW-Only and SW-Only Directory Protocols

The organization of the processor nodes are shown in Figure 1. Each node consists of a standard blocking-load processor with a cache hierarchy, a memory module containing the local portion of the shared memory, a network interface, and finally a local bus connecting the processor, the memory, and the network interface. While a local bus is important to support multiple processors in each node, we consider only one processor per node in our simulations. However, we will discuss the implications of multiple processors per node later in the paper. The processor node is connected to the mesh network by the network interface.

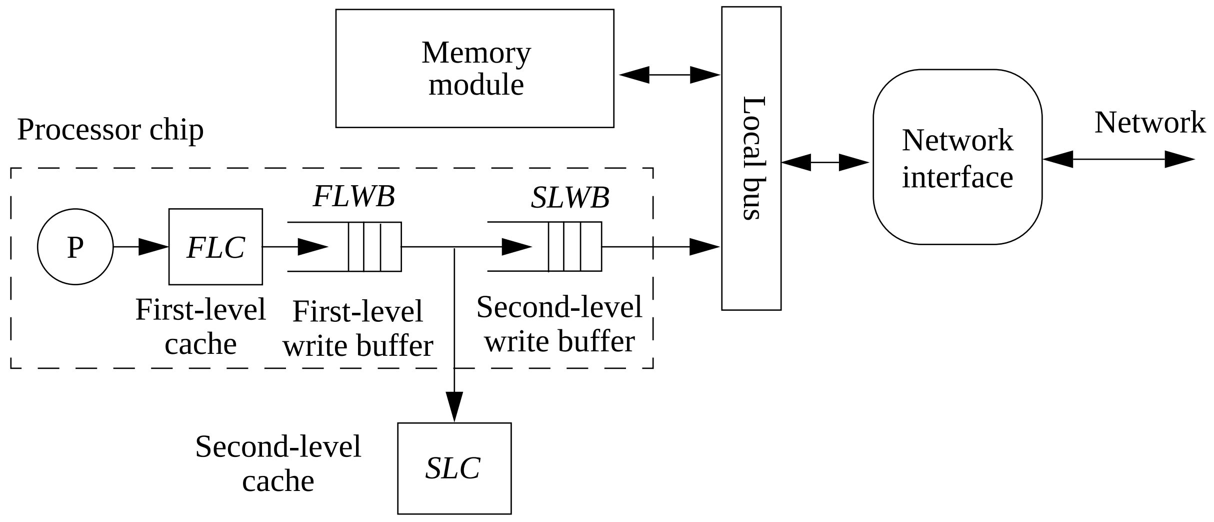

Figure 1: The organization of a processor node.

The two-level cache hierarchy consists of an on-chip write-through first-level cache (*FLC*) and an off-chip copy-back second-level cache (*SLC*) together with first- and second-level write buffers. Both caches are direct-mapped with the same block sizes and full inclusion is supported. Like contemporary microprocessors, e.g., the MIPS R10000 [8] and the PowerPC 620 [9], we assume that the *SLC* controller is on-chip and that there are separate *SLC* and bus interfaces. The cache-protocol engine of the coherence protocol is implemented by the *SLC* controller which also can invalidate blocks in the *FLC* in order to maintain coherence. The *SLC* is lockup-free and keeps track of pending requests by a request buffer denoted *SLWB.* While a lockup-free *SLC* also supports latency tolerating techniques such as relaxed memory consistency models and prefetching, our main motivation in the context of software-only directory protocols is the need for multiple pending requests as we will discuss in Section 2.3.

The network interface is responsible for routing messages between the processor node and the network. Chaiken and Agarwal identified in [5] that counting invalidation acknowledgments in hardware is a key mechanism for software-extended protocols. Therefore, we assume that the network interface collects acknowledgments from remote nodes and notifies the memory-protocol engine when the last acknowledgment to a pending write request has arrived. However, as also Piscitello observed in [17], the handling of invalidation acknowledgments may be troublesome. If the number of invalidations upon a write request is high, it may take a substantial amount of time to issue all invalidation messages. As a result, acknowledgments can arrive before all the invalidations have been issued. The node is capable of handling invalidation acknowledgments even though all invalidations have not been issued yet by incrementing the acknowledgment counter when an invalidation is issued and decrementing it when an acknowledgment arrives.

2.2 Baseline Processor Node for the Hardware-only Directory Protocol

When describing the directory-based write-invalidate protocol used in this study, we will refer to the node where the page containing the block is allocated as *home*; *local* as the node from where a request originated; and finally, *remote* as any other node involved in a coherence action. The protocol assumes two stable states for a memory block: *clean* and *dirty*. In addition, a number of transient states are used to indicate pending protocol transactions. Coherence requests to blocks in transient states have to be retried as in, e.g., DASH [16]. A full-map directory [4] keeps track of which caches having copies of a memory block and the protocol actions are as follows.

Read misses in the *SLC* are sent to home. If the block is clean, home sends a block copy to local and updates the directory with the identity of local. Otherwise, if the block is dirty, home issues a write-back request to remote; remote writes the block back to home; and finally, home forwards a block copy to local and updates the directory. The block is in the transient state ‘busy’ from the point in time when home issues a write-back request until home’s copy is updated.

Processor writes to non-exclusive blocks in the *SLC* result in an ownership request sent to home. Home inspects the directory and sends explicit invalidations to the caches with a block copy. A cache that receives an invalidation invalidates its local copy and returns an acknowledgment to home. Home grants ownership, including an exclusive block copy, to local when all acknowledgments have arrived. The memory copy is in the transient state busy from the point in time home issues invalidation requests until all acknowledgments have been collected.

To implement a memory-protocol engine that supports the protocol actions taken by home requires three main mechanisms, which are shaded in Figure 2: a directory to record all copies of a memory block; a state memory to store the global state of each block; and finally, the memory controller, which is responsible for processing coherence requests to home and requires a significant engineering effort to be correctly designed. By contrast, software-only directory protocols trade lower performance against less hardware complexity. Next, we will study in detail the architectural issues associated with software-only directory protocols.

2.3 Baseline Processor Node for a Software-Only Directory Protocol

In software-only directory protocols as defined by Chaiken and Agarwal [5], the directory and the state of a block are stored in memory and managed by software handlers. More specifically, all three parts of the memory-protocol engine in Figure 2 are now emulated by software handlers executed on the compute processor in response to coherence requests. As a result, the hardware

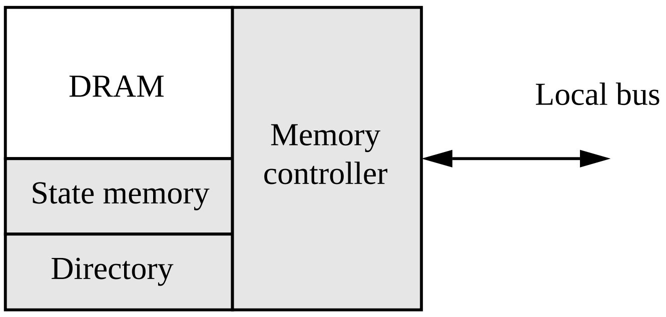

Figure 2: The memory module organization for a hardware-only directory protocol.

cost is reduced and the flexibility of the protocol is significantly enhanced. However, since the compute processors manage the directory, certain requirements must be enforced in the processor node organization. Moreover, to be able to compare cost and complexity of software-only and hardware-only directory protocols careful analysis is required of, e.g., deadlock/livelock situations that may arise. In this section, we pinpoint the major differences in the node organizations and argue that a processor node that supports a software-only directory protocol does not introduce any deadlock situations that can not be circumvented by known techniques which are fairly simple. Figure 3 shows the overall processor node organization for the software-only directory protocol.

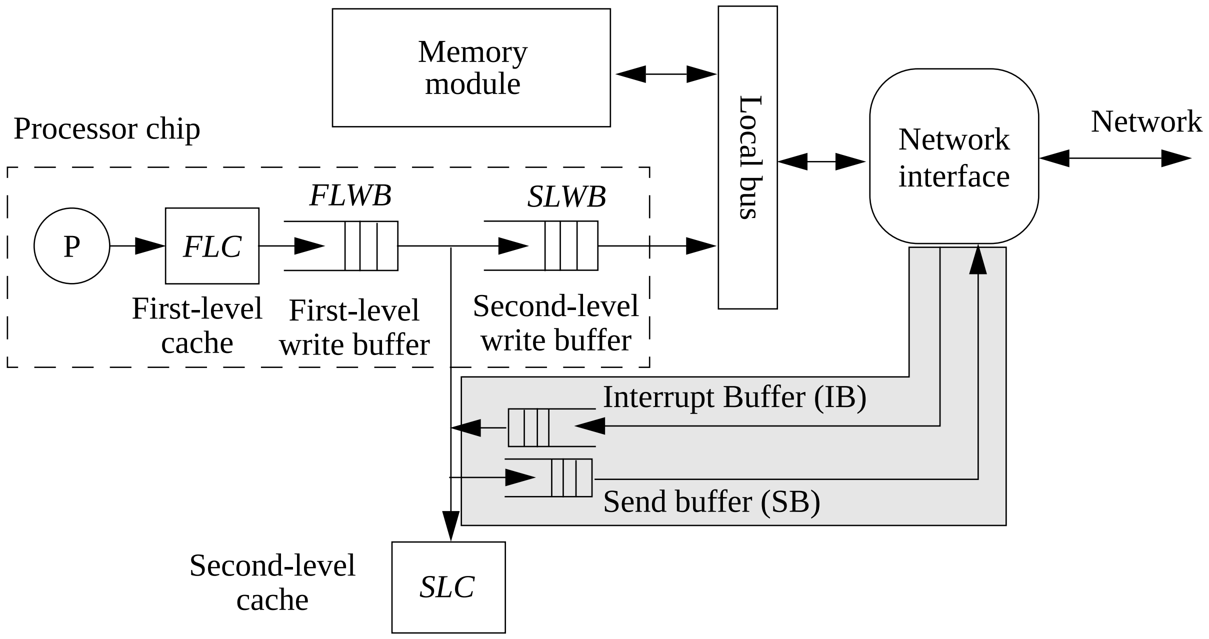

Figure 3: Processor node organization for a software-only directory protocol.

As pointed out in [7], an Interrupt Buffer (IB, shaded in Figure 3) is needed to buffer incoming coherence requests at home. A coherence request arriving at the IB interrupts the compute processor through a low-level interrupt mechanism and a software handler is executed. Active messages [21] rapidly select which software handler to execute. To enable processor initiated message sending, a Send Buffer (SB, shaded in Figure 3) is included. The IB and SB are interfaced to the second-level cache bus as proposed in [12] where the first entry in the buffer (last for SB) is accessible from software. However, we propose to access the buffers through memory-mapped addresses instead of register-mapped as in [12] to adjust to mainstream processor designs.

Coherence interrupts, also called *high-availability interrupts*, need to be precise as pointed out in [14] to avoid *protocol deadlock*; we cannot delay the software handler execution until a pending load completes. To see this, consider two nodes, *i* and *j*, which both encounter a cache miss and the miss from *i* requires service on node *j*, and the miss from *j* requires service on *i*. Then, if the pending load is not interrupted, a circular dependency arises that deadlocks the protocol. A consequence of the need to interrupt the node when a load is pending is that both the first and second-level caches must be lockup-free and capable of handling two cache misses at the same time.

A previously studied livelock issue is that high-availability interrupts open up the *window of vulnerability* [14] which may prevent forward progress of the application. *Associative locking* [14], which is applicable to systems with multiple pending requests and high-availability interrupts, can be used to close the window. The associative locking method locks a block when it is loaded into the cache and defers invalidation of the block until the processor has accessed it, i.e., when the interrupted load instruction has completed.

Another known deadlock issue is related to the fact that a software handler may transmit one or several messages as a result of a coherence request. However, if the send buffer is full when a software handler wants to issue a message, the handler cannot run to completion. As a result, the processor cannot service incoming requests either, which may result in a deadlock situation because of circular dependences between the buffers on different nodes. Note that the same problem arises also in a hardware-only implementation. In, e.g., DASH [16], some request-response dependencies are solved using separate meshes for requests and replies. The general deadlock problem can be solved by augmenting the hardware buffers in software. This technique is referred to as *network overflow recovery* [15]. A time-out mechanism signals a network overflow interrupt to the processor when the send buffer has been full for a predefined amount of time. The processor then moves the contents of the interrupt buffer to a software buffer allocated in memory, thereby freeing buffer space in the hardware. The messages in the memory are rescheduled when the send buffer has drained.

In summary, a processor node that supports a software-only directory protocol requires processors that implement a low-level interrupt mechanism and lockup-free first and second-level caches to interrupt a pending load instruction so as to avoid deadlock. While other deadlock situations show up in this environment, previous studies have demonstrated fairly simple solutions to them. Overall, software-only directory protocols exhibit lower complexity than hardware-only protocols.

3 An Enhanced Processor Node for Software-Only Directory Protocols

We further develop the basic processor node for software-only directory protocols in this section to specifically handle misses and directory accesses more efficiently. When designing the node architecture, we have identified three important performance issues for software-only directory protocols that we specifically address in this section.

1. **The latency of local read misses:** Usually, a cache coherence protocol does not distinguish between shared data and private data. Therefore, also private data, which is most likely allocated in the local portion of the shared memory, incurs the latency for coherence actions even though it is allocated on the local node and is not shared. In a software-only directory protocol this latency may be high and, more importantly, totally unnecessary.
2. **The latency of remote read misses:** The service of misses to remote nodes requires one or several invocations of software handlers in the home node. The software handler latency shows up on the critical memory access path for the miss which may significantly prolong the service of the miss as compared to a hardware-only protocol.
3. **Caching the directory of a memory block:** In a software-only directory protocol, the directory is stored in dynamic RAM. A possible solution to enhance the performance of the software handlers is to cache the directory. However, caching of directory entries may interfere with caching of application data. As a result, a replacement miss occurs which may be to a remote node. Since remote misses incur a severe latency, the performance gain of the directory caching may be reduced, or even worse, result in a performance degradation.

In [7], we did not address the first and third issues. However, we addressed the second issue from a performance point of view but the implementation considerations were not explored in detail. In this section we address the above issues from a performance as well as an architectural point of view.

3.1 Optimization of Read Miss Requests to the Local Node

Ideally, read misses to the memory in the local node, i.e., local is equal to home, should not interrupt the compute processor. As we will see, our choice of a local bus between the processor, the memory, and the network interface (see Figure 3) instead of a central node controller as in, e.g., the FLASH [11] is fundamental in achieving this goal. Another key decision is to assume a separate state memory as in the hardware-only implementation as shown in Figure 4. A memory controller is added that returns a block copy to the local processor as long as the block is clean. If the block is dirty or marked busy, however, the memory controller responds with a reject message which is handled by the network interface. The network interface handles a local read reject as a remote read miss, i.e., it puts the miss request in the IB and interrupts the processor. To assure correctness, the state memory is not allowed to be cached in the local processor caches. Note that the memory controller in Figure 4 is much simpler than the controller in Figure 2 since it only supports a few simple operations and does not implement the whole coherence protocol.

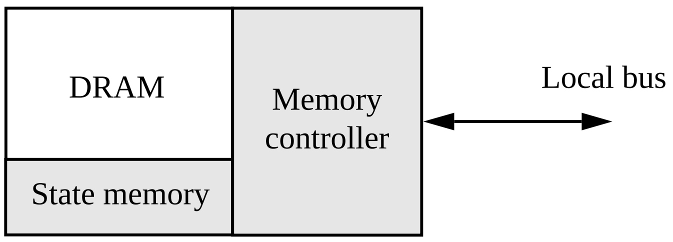

Figure 4: The memory module organization in a software-only directory protocol with support for efficient handling of read misses to local memory.

Since a local read miss to a clean block does not interrupt the compute processor, the directory managed by software handlers is not updated. In Alewife, a special local-bit is used to indicate whether the local cache has a copy of the block or not [5]. We propose to neglect keeping track of whether there is a block copy in the local cache or not. Instead, we rely on snooping of invalidation requests on the bus in order to maintain coherence in the local cache. However, if there are no remote copies that must be invalidated, the software handler must explicitly invalidate the block in the local cache.

3.2 Optimization of Read Miss Requests to Remote Nodes

The latency of misses to remote nodes is increased in software-only directory protocols because software handlers are invoked at the compute processors associated with the home nodes. In [7], we proposed a strategy to remove the software handler latency from the remote miss latency. However, the support mechanisms for the strategy where only partly addressed. In Section 3.2.1. we first review the strategy from [7] and then describe how it can be incorporated in our processor node in Section 3.2.2.

3.2.1 A High-level View of the Optimization Strategy

As described in Section 2.2, a read miss to a clean block requires service of one coherence request at the home node, i.e., one invocation of a software handler. In the baseline software-only directory protocol, the handler is responsible for sending the block copy to local and for updating the directory. As a result, the handler latency ends up on the critical memory access path. We suggested in [7] to support block data transfer in hardware and then let the software handler update the directory. Thus, the directory can be updated in parallel with the data transfer.

For read misses to dirty blocks, two handler invocations at the home node are required: the first one occurs when forwarding the read miss to remote; and the second one occurs when home forwards the copy to local and updates the directory after the block copy has been written back to memory. The first handler invocation can be eliminated by supporting request forwarding in hardware. The second handler invocation can take place in the background by supporting data transfer in hardware as for misses to clean blocks.

3.2.2 Implementation of the Optimization Strategy

Most modifications of the processor node are enhancements of the network interface functionality. Like the optimization of local misses, the optimization of remote misses requires a separate state memory for each block (see Figure 4). The state memory has to be accessible from both the processor and the network interface, and therefore, is not allowed to be cached to ensure correctness. Figure 5 shows a logic overview of the network interface which is one example implementation that we use in this study. The Fetch box is responsible for selecting the next message to handle, the Launch box is responsible for depositing messages after service in the network interface, and the Execute box does the logic operations according to the contents of a message. The other boxes are referred to in the text when necessary. In the rest of the section, we will only consider the functional behavior of the network interface.

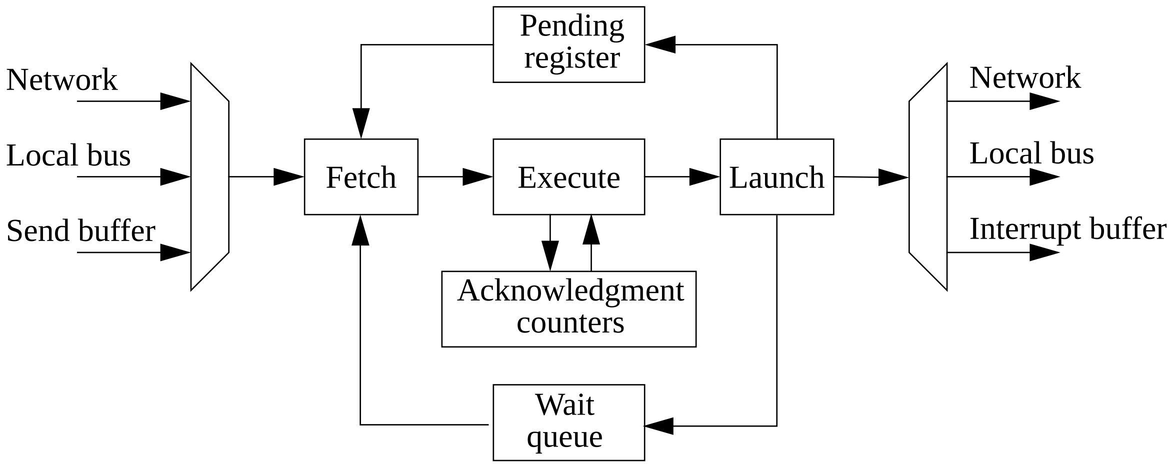

Figure 5: A logic overview of the network interface.

Messages to other nodes are routed through the network interface and to the network. Further, messages to this node in response to a local cache request are routed to the local bus. Finally, all other messages to this node, i.e., coherence messages to home, are handled by the network interface as follows. When a message arrives, the network interface first inspects if it is an invalidation acknowledgment, and thus the corresponding acknowledgment counter is decremented. If the counter reaches zero, i.e., all acknowledgments have arrived, the network interface sends ownership to local (possibly another node) and puts an invalidation acknowledgment in the interrupt buffer so the processor can update the directory. Our simulation results indicate that at most two invalidation counters are allocated simultaneously in more than 90% of the cases and more than 16 counters were never needed. The upper limit depends on the number of processors and the maximum number of pending writes each processor can have.

If the incoming message is not an invalidation acknowledgment, the network interface launches a state-lookup request for the block to the memory module in the node and puts the message in the pending register (see Figure 5). When the memory module returns the state of the block to the network interface, additional data can be supplied. If the block is clean, the memory sends a block copy along with the state information. For dirty blocks, the identity of remote is stored in the first word of the empty block frame, and the memory module sends this word along with the state information. Finally, in all other cases, memory only supplies the block state to the network interface. As will be discussed later, we assure correctness by only allowing one pending state lookup at the time.

When the state of the block is returned from memory, the message is removed from the pending register and rescheduled in the execute box. If the block is marked busy, i.e., there is a pending coherence operation to the block, the network interface sends a retry message to local. If the memory block is not busy, the network interface marks it as busy and takes the following actions depending of the global state of the block. Write requests to clean blocks are inserted in the interrupt buffer since the processor needs to inspect the directory to find out where to send the invalidations. For read requests to clean blocks, the network interface sends a block copy to local and puts the request in the interrupt buffer so the processor can update the directory. In this way, the directory management is done in parallel with the block data transfer to local.

The network interface forwards read and write requests to dirty blocks to remote (whose identity is supplied from memory along with the state information) by issuing a write-back request and then stores the identity of local in the first word of the block frame in memory. When the block copy is written back to home, the network interface forwards a copy to local (whose identity is found in the block frame in memory) and puts a message in the interrupt buffer for directory update. Further, the network interface is responsible for writing the block back to memory. Note that the processor also in this case updates the directory for the block in parallel with the block data transfer. By storing the identities of remote and local in the first word of the empty block frame in memory, the network interface does not need to access the directory. As a result, the directory can be completely managed by software handlers.

We only allow the network interface to do state lookup for one coherence request at the time. Consider two read requests arriving at a dirty block. If we allowed state lookup for both requests simultaneously, they will both find the block dirty and *two* write back requests are issued to remote. Even worse, both read requests store their own identity in the first word of the block frame. As a result, we will loose one of the node identities from which the reads originated. A request going to the processor interrupt buffer marks the memory block as busy for similar reasons. The notion of busy blocks also prevents races between local read misses and remote requests since the local read miss only is served by the memory controller if the block neither is busy nor is dirty. If a new request that requires a state lookup arrives to the node when a state lookup already is pending, the new request is put in a wait queue (see Figure 5). When the previous state lookup completes, the requests in the wait queue have priority over new requests coming from the network, the local bus, and the send buffer. The request from the wait queue is rescheduled and handled by the execute box in the same way as if it came directly from, e.g., the network.

Note that messages that do not require a state lookup by the network interface are still handled even when a state lookup request is pending.

The state machine in the network interface is expected to be less complex than the memory controller of a hardware-only protocol. In a hardware-only protocol the state machine must, e.g., inspect the directory, and based on the value of each entry in the directory either issue an invalidation or not. By contrast, the state machine in the network interface does not manage the directory, and thus can be made simpler.

3.3 Performance Issues when Caching the Directory

Since the latency of the software handler is charged on the compute processor in the home node, a low handler-latency is desirable. One possible solution to reduce the handler execution time is to cache the directory. However, it is an open question if there is any locality in the directory accesses from the software handler. Furthermore, the interaction between application data and directory information has not previously been explored for software-only directory protocols.

When a directory entry is cached, another block — possibly belonging to the application — can be evicted from the cache. A replacement miss occurs if the processor needs the evicted block later. This replacement miss may be to a remote node and may incur a long latency. Thus, a higher access penalty may be encountered by the application program as a result of directory caching. In the worst scenario the application suffers more from the replacement miss than the handler gains from directory caching which may result in a longer execution time. Therefore, an important performance issue to address is how the directory caching impacts the performance of a parallel program.

To build an intuition of the cache behavior when directory caching is done, we develop a simple performance model of a parallel program by assuming that the only penalty is caused by read misses. The total miss penalty of an application can be expressed as:

$$T_{miss} = M_A * MP_A + M_S * MP_S$$

where M_A and M_S is the number of misses for the application and the software handlers, respectively, and MP_A and MP_S represent the average miss penalty for the application and the software handlers, respectively. Since a software handler miss always hits in the local memory, MP_S is significantly lower than MP_A. Therefore, a rather modest increase in M_A resulting from cache conflicts between directory entries and application data may significantly prolong the total execution time of the application. In our experimental section we will quantitatively evaluate these performance effects.

4 Experimental Methodology

In Section 5, we simulate the baseline hardware-only implementation of Section 2.2 and the different software-only directory implementations described in Section 2.3 and 3. We use the

CacheMire Test Bench [3], a program-driven simulator and programming environment. The simulator consists of two parts: a functional simulator of multiple SPARC processors and a memory system simulator. The functional simulator generates memory references which are fed to the memory system simulator, which delays the processors according to its timing model. Thus, the same timing is obtained as in the target system we model and a correct interleaving of events is maintained.

4.1 Simulation Environment and Architectural Parameters

We simulate multiprocessor architectures with 16 processor nodes. The first-level cache (*FLC*) as well as the second-level cache (*SLC*) have 64 bytes blocks and full inclusion is supported, i.e., if a block is present in the *FLC* it is also present in the *SLC*. The default size of the caches are 2 Kbyte for the *FLC* and 64 Kbyte for the *SLC*. Acquires and releases are supported by a queue-based lock mechanism, similar to the one implemented in the DASH multiprocessor [16]. However, in the software-only directory protocols the queue-based lock mechanism is implemented by software handlers. The default page size is 4 Kbyte and the pages are allocated in a round-robin fashion, i.e., pages with consecutive page numbers are allocated to nodes with consecutive identities. Instruction references and references to private data are assumed to always hit in the *FLC*, i.e., they do not incur any memory system latencies. The default memory consistency model in this study is sequential consistency and we block the processor on each shared data access until it has completed.

As for the timing model, we assume SPARC processors and their *FLC*s clocked at 100 MHz. The *SLC* is assumed to be implemented with SRAM technology and its access time is 30 ns for the first word in a block and the remaining words are transferred 16 bytes at the time resulting in a total block access time of 60 ns in the *SLC*. The memory module is assumed to be implemented in DRAM technology with a 16 byte wide interface. The memory access time is 90 ns to the first word and the rest of a block is transferred 16 byte at the time resulting in a total memory access time of 120 ns for a block. The bus in the processor nodes is assumed to be a 128-bit wide split-transaction bus running at 100 MHz. Thus it takes 10 ns to arbitrate the bus and then 10 ns to transfer a request over the bus. If several units on the bus want to use the bus, bus grant is given to a specific unit using a round-robin arbitration scheme. We simulate a very fast bus since the impact of a particular bus arbitration scheme and the bus load is not the target for our analysis. The network interface is assumed to be clocked at 100 MHz which is the same as the target speed of the MAGIC controller in the FLASH multiprocessor [11].

The processor nodes are interconnected by a 4-by-4 wormhole routed synchronous mesh with a flit size of 64 bits. The mesh is clocked at 50 MHz resulting in a fall-through time of 40 ns for each node. The bandwidth into and out of each processor node is 400 Mbytes/s. The latency and bandwidth in the mesh are comparable to the mesh used in the Stanford FLASH [10, 11]. We correctly model contention of all parts in the system. Table 1 shows the time it takes to satisfy a read request from different levels in the memory hierarchy in the hardware-only implementation assuming no contention. However, in our simulations a request usually takes longer time as a result of contention. Clearly, in a software-only directory protocol requests take longer times as a result of software handler invocations.

Table 1: Average read miss latency times for a hardware-only implementation assuming a conflict-free system.

Latency Numbers for Read Requests	1 pclock = 10 ns (100 MHz)
Fill from FLC	1 pclocks
Fill from SLC	6 pclocks
Fill from Local Memory	28 pclocks
Fill from Home (clean, 2-hop)	89 pclocks
Fill from Remote (dirty, 4-hop)	189 pclocks

To achieve a fast software handler start-up, the processor needs a fast interrupt mechanism. It may be implemented either as a conventional interrupt hardware or as a multithreaded processor, e.g., Sparcle [2] which is used in the Alewife [1]. In this study we assume the context switch times from the optimized Sparcle processor described in [2] and use them as a base for how fast a software handler starts to execute. Further, the access times of the interrupt and send buffers are the same as for the *SLC*, i.e., 3 pclocks. The default latency of a software handler is 50 pclocks. In these 50 pclocks the following actions are included: interrupt handling (4 pclocks), reading the message from the IB (6 pclocks), dispatch the associated software handler (4 pclocks), directory management (32 pclocks), and restart of the application program (4 pclocks). In addition, we charge the handler execution time with 13 pclocks to read or write the state of a memory block, 1 or 16 pclocks for a directory access depending on the directory is cached or not, and finally, 6 pclocks for each message the handler sends.

Since we do not address how the memory allocation scheme for the directory is implemented, we assume a very simple mapping from a data address to the corresponding directory address. We associate four bytes of status including directory and state information with each data block, i.e., the status of 16 data blocks is allocated in one memory block. The mapping from a data address to the corresponding status entry locks as follows (in C-syntax):

$$Address_{directory} = \text{0x70000000} \mid ((Address_{data} \,/\, block\_size) * 4)$$

4.2 Benchmark Programs

To evaluate our architectures we use four parallel programs, where three of them (Water, Ocean, and MP3D) are from the SPLASH benchmark suite [20] and the fourth (LU) has been provided to us by Stanford University. The programs are from the scientific and engineering domain and their main characteristics together with the data set sizes we use are shown in Table 2. All applications are written in C and parallelism is expressed using the *parmacs* macros from Argonne National Laboratory. The applications are compiled with `gcc` version 2.1 and optimization level `-O2`. Statistics are gathered in the parallel section of the programs to avoid initialization effects, which we assume to be negligible in an execution with more realistic data sets.

Table 2: Parallel programs and the data set sizes used in our simulations.

Application	Description	Data set size/Input data
Water	Water molecular dynamics simulation	288 molecules, 4 time steps
LU	LU-decomposition of a dense matrix	200x200 matrix
Ocean	Simulate eddy currents in an ocean basin	128-by-128 grid, tolerance 10^{-7}
MP3D	Particle-based wind-tunnel simulator	10,000 particles, 10 time steps

5 Experimental Results

In this section we quantitatively evaluate the performance issues we described in Section 3. In Section 5.1, we present the performance improvement of our optimization for remote misses from Section 3.2, and we address the effects of caching the directory in Section 5.2. In all our simulations, the local miss optimization strategy described in Section 3.1 is applied.

5.1 Performance of Optimizing Remote Misses in Software-Only Directory Protocols

In this section, we evaluate the relative effectiveness between the two baseline systems in Section 2 and the enhanced software-only directory implementation of Section 3 where optimizations of read misses are applied; however, directory caching is not applied. The application execution times are shown in Figure 6. For each application three bars are shown. The first bar corresponds to the execution time of the hardware-only protocol (HW), and the next two bars correspond to the software-only directory protocol without (SW) and with optimizations for local and remote misses (SW+). For each bar, we decompose the execution time (from bottom to top) into the following components: the busy time, the read, the write, and the synchronization stall times; and finally, for software-only directory protocols, the overhead in servicing coherence actions from other processors by software handler execution (P_TIME).

By comparing the performances of HW and SW in Figure 6, we observe that all stall times are increased under SW as compared to HW. As a result, the execution times are between 31% and 161% longer under SW than under HW. This stems from the invocation of software handlers for each coherence action. The software handler execution prolongs both the service time of a read or write request seen by local as well as the execution time of an application by occupying the compute processor. P_TIME may constitute a significant part of the overhead for the application execution. In [7], we found that less that 20% of the handler execution time were hidden by other stall times and this study confirms those results. Another severe limitation of software-only directory protocols concluded in [7], and confirmed in this study, is an increased synchronization over-

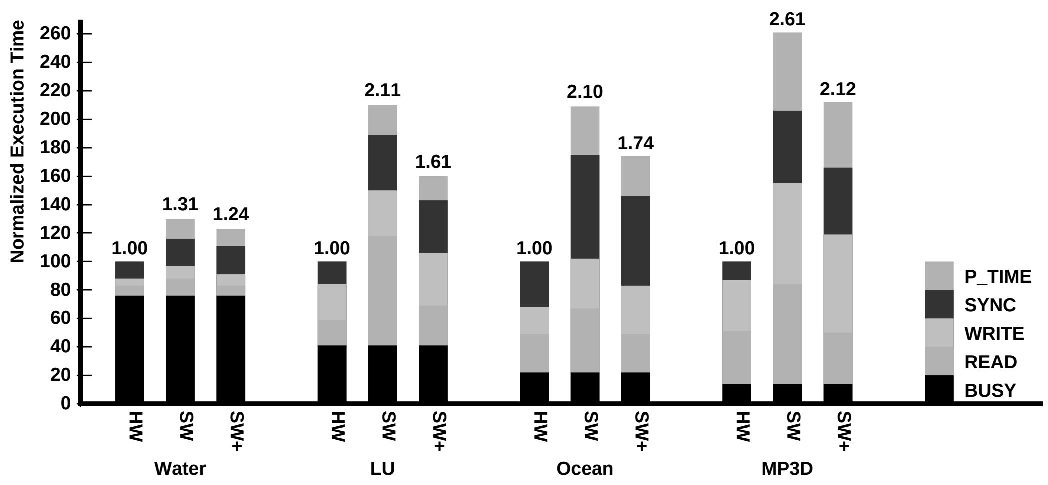

Figure 6: Normalized execution times for the applications under the HW, SW, and SW+ protocols.

head resulting from imbalance between the number of coherence interrupts on the processors. For example, in MP3D the processor with most coherence interrupts encounters ten times as many interrupts as the processor with the fewest interrupts.

Moving on to the execution times under SW+, where we have applied the miss optimizations of Section 3, we find that they are between 5% (Water) and 24% (LU) shorter than the execution times under SW. This stems from a significantly lower read stall time as a result of our optimization of misses. In [7], we concluded that the read stall time of SW+ were virtually the same as for HW and, as can be seen in Figure 6, this conclusion also holds when we consider a detailed model of the processor node as we do in this study. In Figure 6, we also observe that the write stall time is virtually the same for SW and SW+ but significantly longer than for HW. Upon a write request in a software-only directory protocol, the processor must do a directory lookup before it can start to issue invalidations. Therefore, our optimizations from Sections 3.1 and 3.2 do not help reducing the write stall time. The resulting execution times under SW+ is between 24% and 112% longer than under HW.

5.2 Effects of Caching the Directory Information

In this section we evaluate the effects of caching the directory in the processor by considering a software-only directory protocol with the optimizations for misses, i.e., SW+. In Figure 7, we show the resulting execution times for a software-only directory protocol without (SW+) and with (SW+Dir) directory caching normalized to the execution time of a hardware-only protocol (HW).

By comparing the execution times of SW+ and SW+Dir in Figure 7, we find that they are reduced by between 6% (Water) and 19% (MP3D). The reduction stems from a lower handler execution time that reduces the write and synchronization stall times as can be seen in Figure 7. We also note that the overhead for handler execution time that is not hidden (P_TIME) has decreased significantly for all applications. The resulting execution times under SW+Dir is only 16% to 72% longer than under HW. This result is encouraging since MP3D can be considered as

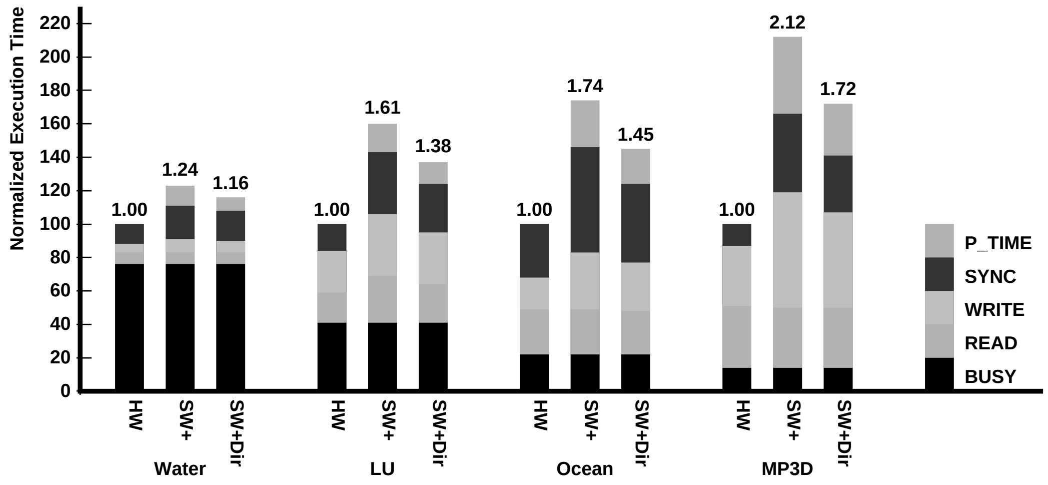

Figure 7: Normalized execution times for the applications with a hardware-only protocol (HW) and a software-only directory protocol with (SW+Dir) and without (SW+) directory caching.

a 'worst-case' application which is known to have low speed-up [16]. We have varied the *SLC* size from 4 Kbyte to infinitely sized *SLC*s and found the results consistent with those for the 64 Kbyte *SLC* size for all applications, i.e., *caching the directory results in shorter execution times*.

To understand the results above, recall the simple execution time formula from Section 3.3:

$$T_{miss} = M_A * MP_A + M_S * MP_S$$

where M_A and MP_A is the number of misses and the miss penalty, respectively, for the application, and M_S and MP_S are the corresponding numbers for the software handlers. Clearly, in a software-only directory protocol without directory caching, M_S is always equal to the number of directory accesses. Since MP_A is significantly higher than MP_S, a small increase in M_A as a result of conflicts in the cache may increase the total execution time of the application even though M_S has decreased significantly. To quantify these effects, we measured the miss rates for the application and the software handler.

The resulting miss rates of the applications and the software handlers are summarized in Table 3. The directory-conflict row in the table is the miss rate the application encounters as a result of cache conflicts between a directory entry and a data block. By comparing the directory-conflict miss rates with the total miss rate the application encounters, we observe that caching the directory only contributes very little to the total application miss rate, e.g., for Water only 0.02% as compared to 0.79%. In addition, we see in Table 3 that the miss rates for the software handlers are only between 0.57% (MP3D) and 4.85% (Ocean). These numbers for the handlers indicate that there is high locality in the accesses to the directory entries. This result is not obvious since accesses to a home node originate from many different nodes.

We have also evaluated the effects of directory caching when the remote miss optimization is not applied, i.e., the SW architecture. In the SW architecture, block data transfer is not supported in hardware so the software handler is responsible for explicit data transmissions. When the handler wants to send a data block, it first accesses (i.e., caches) the block. As a result, the number of

Table 3: Miss rates for the applications and the software handlers assuming 64 Kbyte second-level caches.

		Application			
		Water	LU	Ocean	MP3D
Application miss rates	Cold	0.02%	0.48%	0.02%	0.73%
	Coherence	0.46%	0.07%	0.87%	7.21%
	Replacement	0.29%	0.37%	0.93%	0.94%
	Directory conflict	0.02%	0.02%	0.10%	0.20%
	Total	0.79%	0.94%	1.92%	9.08%
Handler miss rate		0.62%	2.40%	4.85%	0.57%

cache conflicts between the software handler and the application increases as compared to SW+Dir. The total application miss rate under SW is between 0.82% (Water) and 8.81% (MP3D) for 64 Kbyte *SLC*s. Surprisingly, the total miss rate for MP3D is lower under SW than under SW+Dir. When the software handler caches a data block, it becomes clean in home and shared in the cache. As a result, when home accesses the block it is found in the processor cache. MP3D is the only application where the total miss rate for the application is lower under SW with handler caching than under SW+Dir. The total miss rates (Directory + Data) for the software handlers vary between 50.9% (LU) and 92.9% (MP3D). Software handler accesses to data blocks contribute to almost the entire handler miss rates.

In summary, when both the local and remote miss optimizations are applied and the directory information is cached, as in SW+Dir, the execution times are only between 16% and 72% longer than for HW. Further, our results indicate that caching the directory information consistently results in better performance, and that there is very little cache interference between the application data and the directory information. Finally, the miss rates for directory accesses are very low, only between 0.57% and 4.85%, which indicates a high locality in the directory accesses.

6 Discussion and Related Work

Several research projects are headed towards software managed coherence protocols. The research has evolved along two main directions: either a separate protocol processor is used to execute the software handlers that emulate the coherence protocol or the handlers are executed on the compute processor.

The first direction is represented by, e.g., the Stanford FLASH [10, 11] and the Wisconsin Typhoon [18]. These projects suggest using a separate processor to execute the software handlers. The processor can be located in a central node controller as in FLASH or located in the network interface as in Typhoon. Like us, these projects have the goal to achieve flexibility in the protocol design. However, it is an open question whether the performance justifies the cost of an extra processor. In fact, the small performance difference obtained in this study between the hardware-only and the software-only directory protocols indicates that the expected performance gain from a separate protocol processor is low as compared to running the software handlers on the compute processor.

Several design efforts aim at executing the software handlers on the compute processor, e.g., the MIT Alewife [1], the Cooperative Shared Memory [13], and the StarT-NG [6] projects. Both the Alewife and the Cooperative Shared Memory efforts suggest using a hardware protocol that handles a limited number of copies and rely on software handlers only when the hardware directory overflows. In [5], Chaiken and Agarwal evaluate the cost and performance of a number of protocols ranging from protocols with zero hardware pointers, i.e., software-only directory protocol, to a full-map protocol. They suggest that a minimum of one hardware pointer shall be supported in hardware. By contrast, our results indicate that also software-only directory protocols can be competitive with hardware-only protocols.

The StarT-NG project suggests connecting commodity microprocessors by a bus and thus building clusters. The clusters are then connected by a network and coherence between the clusters are maintained through a software-only directory protocol. A hardware unit snoops on the cluster bus and when a processor issues a request to globally shared memory, the bus request is intercepted, a compute processor is interrupted and a software handler is executed to service the request which may trigger handler invocations at other clusters as well. A severe disadvantage in StarT-NG, that we have addressed and solved in this study, is that read miss requests to the local memory incur software handler overhead.

A node architecture with a local bus, such as our proposed architecture as well as StarT-NG, is more suited if we want to build clusters containing several processors per node than, e.g., FLASH and Alewife, which both have a central node controller within each processor node. A central node controller is harder to scale to more than one processor due to pin limitations and other practical considerations.

Our suggested node organization with a local bus is expected to be more efficient than other proposed solutions. In FLASH for example, all local accesses go through the protocol processor and in Start-NG, which also has a node organization with a bus, a read miss to the memory in the local node interrupts the compute processor. By contrast, Alewife can always handle local read misses to clean blocks without software handler overhead, but like in FLASH the miss has to go through a central node controller.

A software-only solution related to our work is Blizzard [19], which provides user-level shared memory on a message passing machine without any hardware support for shared memory. Like our software-only directory protocol, Blizzard invokes software handlers on the compute processors when a load or store to shared memory cannot be satisfied in the local cache. Several implementations of Blizzard are proposed and evaluated. The implementation most related to our work is Blizzard-E, which manipulates the error correcting code at the memory to check if an access can complete or a handler needs to be invoked. For example, upon a write to a read-only block, an exception occurs and a user-level handler is executed. However, their results are difficult to compare with ours since they do not present the performance of Blizzard-E relative to a hardware-only architecture with a similar organization.

7 Conclusions

In software-only directory protocols, i.e., protocols where the directory is allocated in memory and managed by handlers executed on the compute processor, the invocation of software handlers may prolong the service of a read miss request. In a previous study [7] we proposed a strategy to remove the handler latency from the critical memory access path of a miss. However, the implementation considerations were only partly addressed.

In this study we have proposed a processor node architecture with efficient support for software-only directory protocols. We have specifically addressed deadlock issues related to the implementation of software-only directory protocols and three important performance issues for software-only directory protocols: the performance of read misses to local memory, the performance of read misses to remote nodes, and the performance effects of caching the directory when executing a software handler.

By using simulations of a detailed multiprocessor model and four parallel benchmarks from the SPLASH suite [20] we have found that the read latency of remote misses in software-only directory protocols is virtually the same as in a hardware-only protocol when our strategy is applied. Further, by using a separate state memory for each memory block we have shown that read misses to local memory can be serviced without software handler overhead. Finally, we conclude that caching the directory impacts very little on the total miss rate of an application. In addition, caching the directory decreases both the handler execution time and the application execution time for all studied applications and for all evaluated cache sizes (4 Kbyte to infinite size). Overall, a software-only directory protocol with both local and remote miss optimizations and caching of the directory reaches between 58% and 86% of the performance of a hardware-only protocol while demonstrating a design point with less cost and complexity.

Acknowledgments

We would like to thank the people in the RPM project at the University of Southern California and the people in the FLASH project at the Stanford University for their valuable comments after Prof. Stenström’s talks there. This research has been funded in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under contract number P855.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife Machine: Architecture and Performance,” in *Proceedings of 22nd International Symposium on Computer Architecture,* pages 2-13, June 1995.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D’Souza, and M. Parkin, “Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors,” *IEEE Micro,* 13(3):48-61, June 1993.

[3] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, “The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors,” In *Proceedings of the 26th Annual Simulation Symposium,* pages 41-49, March 1993.

[4] L. M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Transaction on Computers*, C-27(12):1112-1118, December 1978.

[5] D. Chaiken and A. Agarwal, “Software-Extended Coherent Shared Memory: Performance and Cost,” In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314-324, April 1994.

[6] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton, “StarT-ng: Delivering Seamless Parallel Computing,” In *Proceedings of EURO-PAR ’95*, Lecture Notes in Computer Science, No. 966, Springer-Verlag, Berlin, pages 101-116, August 1995.

[7] H. Grahn and P. Stenström, “Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors,” In *Proceedings of the 22nd International Symposium on Computer Architecture,* pages 38-47, June 1995.

[8] L. Gwennap, “MIPS R10000 Uses Decoupled Architecture,” *Microprocessor Report,* 8(14):18-22, October 1994.

[9] L. Gwennap, “620 Fills Out PowerPC Product Line,” *Microprocessor Report*, 8(14):12-16, October 1994.

[10] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, “Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor,” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38-50, October, 1994.

[11] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor,” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI),* pages 274-285, October, 1994.

[12] D. S. Henry and C. F. Joerg, “A Tightly-Coupled Processor-Network Interface,” In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pages 111-122, October, 1992.

[13] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, “Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,” *ACM Transactions on Computer Systems,* 11(4):300-318, November 1993.

[14] J. Kubiatowicz, D. Chaiken, and A. Agarwal, “Closing the Window of Vulnerability in Multiphase Memory Transactions,” In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 274-284, October 1992.

[15] J. Kubiatowicz and A. Agarwal, “Anatomy of a Message in the Alewife Multiprocessor,” In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 195-206, July 1993.

[16] 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):41-61, January 1993.

[17] J. Piscitello, “A Software Cache Coherence Protocol for Alewife,” Master’s Thesis, Department of Electrical Engineering and Computer Science, MIT, May 1993.

[18] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: User-Level Shared-Memory,” In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325-336, April 1994.

[19] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhart, J. R. Larus, and D. A. Wood, “Fine-grain Access Control for Distributed Shared Memory,” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297-306, October, 1994.

[20] J-P. Singh, W-D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared-memory,” *Computer Architecture News*, 20(1):5-44, March 1992.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: a Mechanism for Integrated Communication and Computation,” In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256-266, May 1992.