In Proceedings of the 22nd Annual International
Symposium on Computer Architecture, pages 38-47, June 1995

Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors

Håkan Grahn and Per Stenström

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden
Internet: {nesse,per}@dit.lth.se, http://www.dit.lth.se/cachemire/

Abstract

The cost, complexity, and inflexibility of hardware-based directory protocols motivate us to study the performance implications of protocols that emulate directory management using software handlers executed on the compute processors. An important performance limitation of such software-only protocols is that software latency associated with directory management ends up on the critical memory access path for read miss transactions. We propose five strategies that support efficient data transfers in hardware whereas directory management is handled at a slower pace in the background by software handlers. Simulations show that this approach can remove the directory-management latency from the memory access path. Whereas the directory is managed in software, the hardware mechanisms must access the memory state in order to enable data transfers at a high speed. Overall, our strategies reach between 60% and 86% of the hardware-based protocol performance.

1 Introduction

Private caches in conjunction with a directory-based cache coherence protocol have been used in shared-memory multiprocessor designs such as the Stanford DASH [14] to achieve high performance. The mechanisms needed to support a directory-based protocol include a protocol engine in each cache and in each memory module. Associated with the protocol engine in the memory module is a directory that keeps track of each copy of a shared memory block. As demonstrated by the DASH design, the hardware cost and complexity of this memory-protocol engine and the associated directory contribute significantly to the total logic overhead in a processing node. Furthermore, since the protocol is hard-wired, flexibility in its design and optimization is not provided. Recent efforts have addressed these concerns by migrating the entire protocol-engine functionality, or parts of it, to software handlers executed on a microprocessor.

In the Stanford FLASH [10] and the Wisconsin Typhoon [15] design efforts, the approach is to emulate the memory-protocol engine by software handlers on a dedicated protocol processor. In FLASH, for example, a special-purpose protocol processor called MAGIC is designed to support a range of protocols including message-passing primitives. Clearly, flexibility is met whereas the introduction of an extra processor apparently is costly. A more radical approach has been taken in the MIT Alewife machine [1] and in Cooperative Shared Memory [12] where the memory-protocol engine uses a limited directory organization with a fixed number of hardware pointers per memory block. When a pointer overflow occurs, the compute processor extends the functionality of the protocol to a full-map protocol by handling coherence actions by software handlers [5]; hence, this protocol class is called *software-extended protocols* [6].

Chaiken and Agarwal [6] evaluate a spectrum of software-extended protocols with respect to performance and cost ranging from *software-only protocols* that emulate directory management entirely in software on the compute processor to protocols with up to five pointers in hardware which means that the software handler is only executed when more than five copies exist. By comparing the performances of these protocols with a hardware-only implementation of a full-map protocol they find that the single hardware-pointer scheme does comparably well with the hardware-only protocol; it reaches between 42% and 100% of the performance of the hardware-only protocol. On the other hand, while the software-only protocol is the cheapest solution, they show that its performance is significantly lower than the single pointer-scheme.

In this paper, we study the reasons for performance limitations of software-only protocols and identify in the process several strategies that can enhance their performances significantly in the context of cache-coherent NUMA architectures with write-invalidate protocols. We have found that when a processor triggers a miss or an ownership acquisition, the latency these transactions involve are dominated by the software latency associated with directory management. In other words, the software emulation of the directory appears on the critical memory access path. To remove this devastating latency component, we propose to overlap it with the inter-node transaction latencies. Our approach is to use efficient hardware-based data transfer mechanisms and then let the compute processor handle the directory at a slower pace in the background.

In order to justify our approach from a performance perspective, we compare the performance of our software-only protocol strategies with that of a hardware-only protocol using architectural simulations and four parallel applications. We find that for two of the applications, the overall effect of our strategies is that the software-only protocol performance is more than 85% of the hardware-only protocol performance. Whereas the performance for one of the applications (MP3D from SPLASH [16]) is only 60% of that of the hardware-only protocol, this application suffers from a devastating miss rate for the hardware-only protocol as well. We show that our approach of overlapping directory management with miss handling effectively removes the software handler execution time from the critical memory access path for read-miss transactions. Moreover, we also find that relaxing the memory consistency model is as effective for software-only protocols as for hardware-

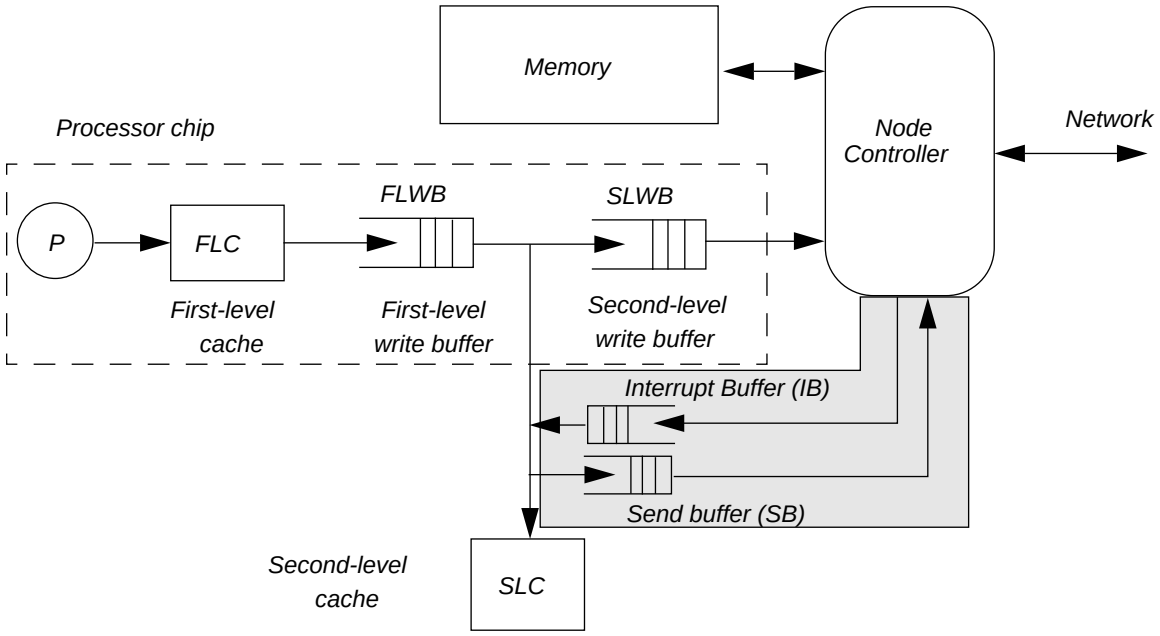

Figure 1: Organization of each processor node. The shaded areas only apply to the software-only protocol that is discussed in Section 2.3.

only protocols to hide the latency associated with ownership acquisition.

As for the rest of the paper, we present in the next section our architectural framework and the baseline hardware-only and software-only protocols we simulate. This also serves as a motivation for the improvement strategies of the software-only directory protocols we propose in Section 3. In Sections 4 and 5, we present our experimental methodology and results from our simulations. Finally, we conclude the study in Section 6.

2 HW-Only and SW-Only Directory Protocols

Our architectural framework consists of a cache-coherent non-uniform memory access (CC-NUMA) architecture with a write-invalidate directory cache coherence protocol. We start in Section 2.1 by describing the architectural framework and the write-invalidate protocol we use in our simulations. This framework is identical for all our simulated hardware-only and software-only protocols. We continue in Sections 2.2 and 2.3 with the specific functionality of the hardware-only and of the software-only directory protocols, respectively, which are used as baselines in the rest of the paper. We only discuss the protocol engines associated with the memory modules; the cache-protocol engines are identical for the two types of protocols.

2.1 The Baseline Architecture and Cache Protocol

The processor nodes, whose organization appears in Figure 1, are interconnected by a network. A processor node consists of a processor with a two-level cache hierarchy and associated write buffers with a similar organization as many contemporary microprocessors such as the MIPS R10000 and the PowerPC 620. The cache hierarchy is interfaced to the local portion of the shared memory and the network by a node controller according to Figure 1. The first-level cache (*FLC*) is a write-through on-chip cache whereas the second-level cache (*SLC*) is a copy-back cache. In order to support relaxed memory consistency and prefetching, whose effects we study to some extent, we assume that the *SLC* is lockup-free. Both caches are direct-mapped and have the same line sizes. Thus full inclusion is supported; if a block is present in the *FLC* it is also present in the *SLC*. The *SLC* controller is on-chip as in, e.g., the MIPS R10000. The *SLC* controller implements the cache-protocol engine of the write-invalidate coherence protocol and can invalidate a block in the *FLC* in order to maintain consistency. The processor node supports any memory consistency model. By default, we assume sequential consistency and blocks the processor on each shared data access until it has completed. We will also show results under release consistency [8] and will then assume the full functionality of the lockup-free capability of the *SLC*.

The task of the node controller in Figure 1 is to route messages to and from the network and inside the node. Regardless of whether the directory is managed by hardware or software mechanisms, the node controller is also responsible for collecting invalidation acknowledgments and notifying the memory-protocol engine when the last acknowledgment has arrived. We do this because it was identified as a key mechanism for software-extended protocols in [6] which would otherwise result in as many interrupts as the number of acknowledgments.

When describing the write-invalidate protocol, we will refer to the node where the page containing the block is allocated as *home*; *local* as the node from which the request originated; and finally, *remote* as any other node involved in a coherence action. In our write-invalidate protocol, the stable states of a memory block are *clean* and *dirty*. The memory block can also be in a transient state as a result of pending protocol transactions. Read-miss and ownership requests to transient memory blocks are forced to be retried as in the DASH [14]. Moreover, a directory keeps track of the copies of the memory block. The protocol actions, which are similar to those of the Censier and Feautrier protocol [4], are as follows.

Processor reads that miss in the *SLC* are sent to home. If the memory block is clean, the miss is immediately serviced by home which records local in the directory. By contrast, if the block is

dirty, home forwards the request to remote; remote writes the block back to home; and finally, home returns the block to local resulting in as many as four cache-memory transactions to service the read miss if local, home, and remote are different nodes. During the time home has forwarded the request to remote until the block is written back to home, the block is in a transient state and marked as busy. Then, the block becomes clean.

A processor write to an *SLC* copy that is not exclusive forces an ownership request to be sent to home. Home sends explicit invalidations to each cache with a block copy according to the contents of the directory. When a cache gets an invalidation, it invalidates its local copy of the block and sends an acknowledgment back to home. When home has received all acknowledgments, it grants ownership to local. The block is in a transient state and marked as busy when invalidations are pending until all acknowledgments have arrived at home after which the block becomes dirty.

2.2 Mechanisms for a Hardware-Only Protocol

We now review the mechanisms a hardware-only implementation must provide in the memory-protocol engine to support the write-invalidate directory protocol presented in the previous section. The memory-protocol engine is located at the memory in Figure 1 and interfaces to the network and to the local cache via the node controller.

The memory-protocol engine conceptually consists of three parts: the directory, the state memory, and the memory controller. Although other directory organizations are possible in a hardware-only directory protocol, we simulate a full-map protocol implemented by presence-flag vectors according to Censier and Feautrier [4] that contain N bits for each memory block, assuming N nodes. The state memory encodes the state of each memory block and in our simulated hardware-only protocol three bits are associated with each block.

Finally, the memory controller implements the coherence protocol actions of the memory-protocol engine. This controller typically processes an incoming request by carrying out the following tasks: it decodes the message; it performs a directory and state-lookup; it composes new messages; and it sends messages to other nodes. In reality, a fair engineering effort is required to correctly implement the controller and ascertain that all corner cases are covered. In addition, the hardware resources to implement the memory-protocol engine can be quite substantive; in the DASH prototype, for example, the logic overhead of the memory-protocol engine is about 20% for a similar hardware-only directory protocol [14]. This motivates us to migrate the directory management to software handlers which we study next.

2.3 Mechanisms for a Software-Only Protocol

Software-only protocols as introduced by Chaiken and Agarwal [6] refer to a class of protocols where all coherence requests incoming to home are carried out by software handlers executing on the compute processor. More specifically, using the notation in [6] we focus on $Dir_nH_0SW_{NB,LACK}$ where Dir_n means that the system can keep track of n (the number of nodes) copies of a memory block; H_0 means that there are no directory pointers maintained in hardware; finally, $SW_{NB,LACK}$ means a software-extended protocol with no broadcast on directory overflow and where invalidation acknowledgments are collected by a hardware counter and the last acknowledgment interrupts the processor. This extreme variation of a software-extended protocol is particularly attractive from a hardware viewpoint because the whole memory-protocol engine discussed in the previous section is now emulated by software handlers.

We now review how the software-only protocol fits in the architectural framework by considering the basic processor node organization in Figure 1. Since all coherence requests are taken care of by software handlers on the compute processor, the directory and state bits for each memory block discussed for the hardware-only protocol in the previous section can now be allocated in the memory. More importantly, the hard-wired memory controller for the hardware-only protocol is replaced by software handlers.

To enable the emulation of the protocol actions, however, requires some functionality and organizational modifications as compared to a hardware-only protocol. First, the processor must support a low-level interrupt mechanism that can rapidly switch between program execution and protocol execution. Second, an Interrupt Buffer (IB) (shaded in Figure 1) is needed to buffer incoming coherence request. When a coherence request is present, the compute processor is interrupted and executes the corresponding software handler. Active messages [18] can be used to rapidly select which coherence routine the processor will execute. IB is interfaced to the second-level cache bus as proposed in [11], i.e., it has the same access time as the second-level cache and the first request in IB is accessible through memory-mapped addresses. In addition, a Send Buffer (SB) that is interfaced to the second-level cache bus in the same way as IB allows the processor to efficiently send messages in response to coherence requests.

Another issue is that the need for high-availability interrupts in a software-only protocol opens up the window of vulnerability [13]. At the time a block is loaded into the cache as a result of a cache miss, the processor may be busy executing protocol actions. As a result, there is a time window in which the cache block might be invalidated before the processor starts accessing it which may trigger a livelock situation that prevents forward progress. Several methods to close this window are proposed in [13]. We have not addressed this issue but note that it must be done in a real implementation.

In summary, we note that the memory-protocol engine can now be managed in software, and thus there is a potential to avoid a complex, hard-wired memory-protocol engine. However, processors must support a fast mechanism to switch between normal execution and software handler execution. In addition, to justify the software-only approach, the performance must not be significantly lower than the performance of a hardware-only protocol. Although a software-only protocol as presented in this section incurs the overhead of the protocol execution, we will in the next section develop intuition how we can remove parts of it by devoting hardware resources to performance-critical operations.

3 Improving Software-Only Protocols

We start in Section 3.1 with a simple performance model of how coherence actions are handled in the baseline software-only protocol. This serves as a motivation for a range of optimization strategies that we propose in Sections 3.2 through 3.6. The strategies we consider assume various degrees of hardware support and our purpose is to identify what support that makes most sense performance-wise.

3.1 A Simple Performance Model

To build intuition as to what limits the performance of a software-only protocol, we will develop a simple performance model of the execution time of a parallel program in the context of the baseline software-only protocol. Besides the time each processor spends executing the application code, the *busy time*, the processor is sometimes stalled because of read misses, ownership acquisitions, and synchronization operations, referred to as *read stall time*, *write stall time*, and *synchronization stall time*, respectively. These stall time components also show up in a hardware-only protocol but are longer in a software-only protocol. Moreover, the execution time will also be longer in a software-only protocol due to the software-handler execution of requests coming from other processor nodes. However, let us first concentrate on the stall time components.

In Figure 2 we show the latencies involved in a read-miss transaction to a dirty memory block in a scenario where local, home, and remote are different nodes. A read miss to a dirty block incurs the latency of four network hops (*Net*) plus the latency of two handler executions at the home node (*Prot1* and *Prot2*) plus the time to retrieve the block from the remote cache (*C2*). By contrast, the times corresponding to *Prot1* and *Prot2* in the memory-protocol engine of the hardware-only protocol are negligible. Clearly, if the number of handler invocations can be reduced or if they can be carried out in parallel with the message transmission over the network, *Prot1* and *Prot2* can be removed from the critical memory-access path. The strategies presented in Sections 3.2-3.6 aim at doing so.

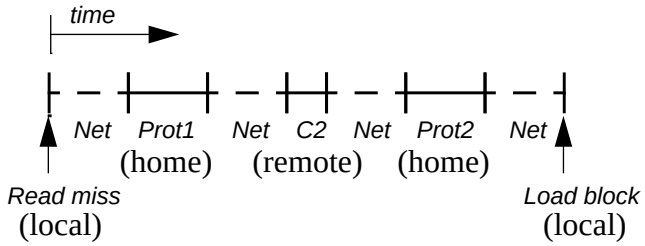

Figure 2: Timing of a read miss to a dirty memory block.

Like the read stall time, the write stall time incurs similar components as in Figure 2. However, since the processor in the home node has to do a directory lookup in order to send invalidations, it is more difficult to remove *Prot1* and *Prot2* from the critical path. Therefore, other techniques to cut the write stall time have to be used and we will specifically study the effects of relaxed memory consistency models.

Even if it is possible to remove *Prot1* and *Prot2* from the memory access path of all memory accesses, they can show up as overhead on the compute processors. However, protocol execution can potentially be hidden if the compute processor is only interrupted when it is stalled due to a read miss, an ownership, or a synchronization request. We will study in detail to what extent this happens in detail in Section 5.

3.2 State-Memory Lookup

Independent of the coherence action at home, the software handler always does a state lookup in memory. Moreover, when a memory block is busy, which can be derived directly from the state bits, the sole task of the software handler is to return a retry message. By supporting these common case operations in the node controller, *Prot1* and *Prot2* in Figure 2 can be reduced by the time of state memory access and, in case of busy memory blocks, by the time to send a retry message. Moreover, as we will see in the following sections, this strategy enables other strategies.

The extra functionality needed for the state lookup strategy is as follows. The node controller must have access to the state of a memory block. An approach to simplify the memory mapping of the memory-block states is to use a separate state memory besides the main memory. To check whether a block is in a transient state, the node controller only needs to look at one bit in the state of a block, the *busy-bit*, given that the states are coded in an appropriate way. Since the node controller already routes and sends messages, it can send retries with virtually no extra functionality. We refer to this strategy as ***SW-1***.

3.3 Reducing Software Latency of Dirty Misses

Once we have incorporated an explicit state memory and the functionality to decode the state bits as in the previous section, further enhancements of the node controller can have dramatic effects on the processor stall time components. One strategy is to let the node controller forward read and write requests to dirty memory blocks directly to remote without interrupting the processor which potentially removes *Prot1* in Figure 2 from the stall time incurred by dirty misses.

To support this strategy, the node controller must be capable of reading and writing the first word in a block frame in memory and change the state bits. When forwarding a request, the node controller inspects the clean/dirty state bit. Moreover, since the memory block frame is unused when a block is dirty, the identity of remote can be stored in the first word of the memory block. After having read the identity of remote, the node controller issues a write back request to remote; stores the identity of local in the first word of the block frame in memory; and finally, sets the busy-bit for the block. In the rest of the paper we refer to this strategy as ***SW-2***.

3.4 Hiding Software Latency of Clean Misses

Misses to clean blocks that reside in another node than local incur two network traversals plus handler execution time (*Net + Prot1 + Net* in Figure 2). Once we let the node controller perform state lookup, it is possible to remove *Prot1* from the critical memory access path by letting the node controller directly send the block to local if the state of the block is clean; the processor interrupt to update the directory can occur in parallel with the second network traversal. This is a natural enhancement of SW-2 since we now have removed *Prot1* for both misses to clean and to dirty blocks. Of course, unlike the SW-2 strategy for removing *Prot1* for dirty blocks, *Prot1* will still be charged to the home processor and can show up as protocol execution overhead for misses to clean blocks.

The functionality enhancements apart from those of SW-2, is the ability for the node controller to read the block from memory. A new node controller action is also needed that involves first sending the block to local and then putting a message in the processor interrupt buffer. When a read miss from local to a clean memory block arrives at home, the node controller first does a state lookup; inspects the clean/dirty state bit; reads the block from memory; sends the block to local; and finally, puts a message into the interrupt buffer. When the processor is interrupted, it only has to update the directory with the identity of local. This strategy is referred to as ***SW-3***.

3.5 Hiding Software Latency of Dirty Misses

Whereas SW-2 and SW-3 can remove *Prot1* in Figure 2 from the critical memory access path for misses to dirty and clean blocks, the next strategy aims at removing *Prot2* from the read stall time and write stall time to dirty blocks. This strategy is built on top of SW-2 and lets the node controller send the block directly to local when remote writes it back to home while interrupting the processor for directory update in parallel with the return of the block to local. In addition to the functionality needed for the SW-2 strategy, this strategy needs a node controller action aiming at first sending the block to local before the processor is interrupted. Throughout the paper we refer to this strategy as ***SW-4***.

3.6 Combining All Optimization Strategies

The most aggressive strategy is to combine SW-1 through SW-4 which potentially removes or hides *Prot1* and *Prot2* for misses to clean as well as dirty blocks. The node controller functionality needed is not beyond what is needed by the other strategies. This combined strategy is referred to as ***SW-5***.

In summary, we propose to apply a range of optimization strategies to the baseline software-only protocol. All strategies address how common case operations can be done in hardware while still letting the directory be managed at a slower pace in software. The most aggressive strategy assumes that the node controller can inspect the individual state bits of the memory block and can forward messages based on the state bits before interrupting the compute processor so as to overlap software latency with network latency. Table 1 summarizes the node controller functionality and expected performance effects for each strategy.

Table 1: Optimization strategies for software-only directory protocols.

Strategy	Functionality in the node controller	Expected performance enhancement
SW-1	Memory state lookup by node controller as well as processor	Can slightly reduce *Prot1* and *Prot2*
SW-2	Forwarding of requests to dirty blocks which requires state-memory modifications and access to first word in a block	Can remove *Prot1* for dirty misses
SW-3	SW-2 plus return of block to local on clean misses before the processor is interrupted which requires access to the memory block	Same as SW-2 but can also hide *Prot1* for clean misses
SW-4	SW-2 plus return of block or ownership to local on dirty misses before the processor is interrupted	Same as SW-2 but can also hide *Prot2* for dirty misses
SW-5	Combination of SW-1 to SW-4	Can remove or hide *Prot1* and *Prot2* for all misses

4 Simulation Methodology

This section presents the simulation framework used to study the relative performance of the protocols. In Section 4.1 we present the simulation environment and the detailed architectural assumptions, and in Section 4.2 we describe the benchmark programs.

4.1 Architectural Parameters

The simulation models are built on top of the CacheMire Test Bench [3]; a program-driven simulation platform and programming environment. The simulator consists of two parts: a functional simulator of multiple SPARC processors and an architectural simulator. The functional simulator issues memory references, and the architectural simulator delays the processors according to its timing model. Thus, the same interleaving of memory references is obtained as in the target systems we model.

We simulate a system of 16 nodes according to Figure 1. The two-level cache hierarchy consists of a 2 Kbyte first-level cache (*FLC*) and an infinite second-level cache (*SLC*) by default, but we will also consider finite *SLCs*. Both caches use 64 bytes blocks. The write buffers contain 16 entries each but under sequential consistency, which is our default consistency model, the processor blocks on each shared reference until it is completed. Acquire and release requests are supported by a queue-based lock mechanism similar to the one implemented in the DASH multiprocessor [14]. The page size is 4 Kbytes and the pages are allocated to memory modules in a round-robin fashion; pages with consecutive page numbers are allocated to nodes with consecutive node 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 timing model assumes that the SPARC processors and their *FLCs* are clocked at 100 MHz (1 pclock = 10 ns) by default. The *SLC* is assumed to be implemented by static RAM with an access time of 30 ns. The processor is assumed to have a 16 bytes wide interface to the *SLC*. The *SLC* is interleaved and assumed to deliver the first 16 bytes of a block after 30 ns and the rest of the block after another 30 ns (16 bytes each pclock). The memory is also interleaved and has a 16 bytes wide interface and assumed to be implemented by dynamic RAM with an access time of 90 ns; the first 16 bytes are supplied after 90 ns and the remainder of the block is transferred 16 bytes each cycle resulting in a total access time of 120 ns for a block. The *SLC* with its write buffer, the memory, and the network are connected by the node controller which is clocked at 100 MHz, which is the same as the target speed of the MAGIC controller in the Stanford FLASH [10]. The only difference between the node controllers for the hardware-only and the software-only protocols are the functionalities summarized in Table 1. In a real system special precaution must be taken to handle deadlocks. However, such mechanisms are similar in both the hardware-only and the software-only protocols and we do not address them in this study; simulations assume infinite buffers.

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 Stanford FLASH [9, 10]. We correctly model contention of all parts in the system. Table 2 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.

A critical timing assumption is how many cycles we charge for the software handlers. First, to achieve a fast start-up of the software handlers, the processor must have a fast interrupt mechanism. It can be implemented either as conventional interrupt hardware in the processor or as a multithreaded processor, e.g., Sparcle [2] which is used in the MIT Alewife [1]. In this study we take the

latency numbers from the optimized Sparcle processor described in [2] and use them as a base for how fast a software handler can be dispatched. Conservatively, the state and directory information of a block are not cached in the simulations. To partly compensate for the slow access to the directory and state memory, we simulate a direct path between the memory and the processor; directory accesses do not go through the node controller. For SW-1 to SW-5 we also assume that a dedicated state memory is supported.

Table 2: Average latency numbers for the hardware-only implementation assuming a conflict-free system.

Latency Numbers for Read Requests	1 pclock=10 ns (100 MHz)
Fill from FLC	1 pclock
Fill from SLC	6 pclocks
Fill from Local Memory	33 pclocks
Fill from Home (2-hop)	79 pclocks
Fill from Remote (4-hop)	169 pclocks

While we also vary the handler execution times in Section 5, we assume 50 pclocks as the default protocol execution time which does not include the time to access the directory and state information. Rather, we include in these 50 pclocks the following actions: interrupt handling (4 pclocks); reading the message (6 pclocks); dispatch of the corresponding software handler (4 pclocks); administration of the software directory (32 pclocks); and restart of the application program (4 pclocks). On top of this we charge 90 ns to read or write the global state of a memory block; 120 ns to read or write a memory block; and finally, 6 pclocks to send a message depending on the type of coherence action. For example, for a read miss to a clean block, 9 pclocks (read the state) plus 12 pclocks (read the block) plus 6 pclocks (send the reply) plus 9 pclocks (update the directory) are added to the basic 50 pclocks, resulting in 86 pclocks to service a read miss to a clean block.

4.2 Benchmark Programs

In order to understand the relative performance of hardware-only and the variations of the software-only protocols, we use four scientific applications written in C using the ANL macros and compiled by `gcc` (version 2.1) with optimization level -O2. Three of them (Water, Ocean, and MP3D) are taken from the SPLASH suite [16] and the fourth (LU) has been provided to us from Stanford University. The applications chosen all have different characteristics. Water has a decent processor utilization whereas MP3D has poor speedup due to a high communication overhead. In LU cold misses dominate, while coherence misses dominate in the other applications. Finally, Ocean has producer-consumer sharing, a high synchronization overhead, and a significant amount of false sharing. Water was run with 288 molecules for 4 time steps. LU used a 200x200 matrix and Ocean used a 128x128 grid with the tolerance factor set to 10-7. Finally, MP3D was run with 10,000 particles during 10 time steps.

5 Experimental Results

In this section we present our experimental results starting with the effects of the different strategies for software-only directory protocols under sequential consistency in Section 5.1. In Section 5.2, we analyze in detail what limits further improvements. The remainder of this section studies architectural variations such as the effects of relaxed memory consistency models (Section 5.3), finite caches (Section 5.4), and finally, software handler execution time variations and the effects of increased processor speed in Section 5.5.

5.1 Efficiency of Software-Only Protocol Strategies

In this section, we evaluate the relative effectiveness of the different strategies for software-only directory protocols proposed in Section 3 by comparing their performances with the baseline hardware-only and software-only protocols in Section 2. In Figure 3, we show the relative execution times of the different optimization strategies. All execution times are normalized to the execution time of the hardware-only protocol. For each application seven bars are shown. The first two bars to the left correspond to the baseline hardware-only (HW) and software-only (BASIC-SW) protocols in Sections 2.2 and 2.3, respectively. They are followed by the strategies listed in Table 1 in Section 3.6. For each bar, we decompose the execution time from bottom to top into the components introduced in Section 3: the busy time, the read, the write, and the synchronization stall times; and finally, for software-only protocols, the overhead in servicing coherence actions by software-handler execution from other processors (P_TIME).

Let us first compare the baseline protocols. As expected, BASIC-SW does significantly worse than HW; the execution times are between 26% (Water) and 151% (MP3D) longer than under HW. From the execution time breakdown, we can see that all stall time components are longer under BASIC-SW which confirms the intuition from Section 3; the software latency is on the critical memory access path which makes the read and write stall times longer. The applications we use have very different miss rates as can be seen from Table 3 that shows the cold, coherence, and total read-miss rates assuming infinite *SLCs.* Since the applications differ in the number of misses, we also see a big difference in the relative performance; while Water has few misses (0.50%) and requires few software handler invocations, protocol handling dominates in MP3D which has a high miss rate (7.83%) and a low processor utilization under BASIC-SW as well as under HW. Next we study how the stall time components are affected when applying the strategies proposed in Section 3.

Table 3: Application read-miss rates under SW for infinite and 16 Kbyte *SLCs*.

	Application			
	Water	**LU**	**Ocean**	**MP3D**
Cold miss rate	0.02%	0.48%	0.01%	0.73%
Coherence miss rate	0.48%	0.07%	0.83%	7.10%
Replacement miss rate (16 Kbytes *SLCs*)	0.87%	1.64%	3.26%	2.45%
Total miss rate (infinite *SLCs*)	0.50%	0.55%	0.84%	7.83%
Total miss rate (16 Kbytes *SLCs*)	1.37%	2.19%	4.10%	10.3%

Starting with SW-1 (as defined in Table 1), in which state lookup is performed by the node controller, we would expect all stall time components and P_TIME to be reduced. Although the improvement is in the modest range of 5% to 10%, we can see that the execution time under SW-1 is slightly shorter than under BASIC-SW.

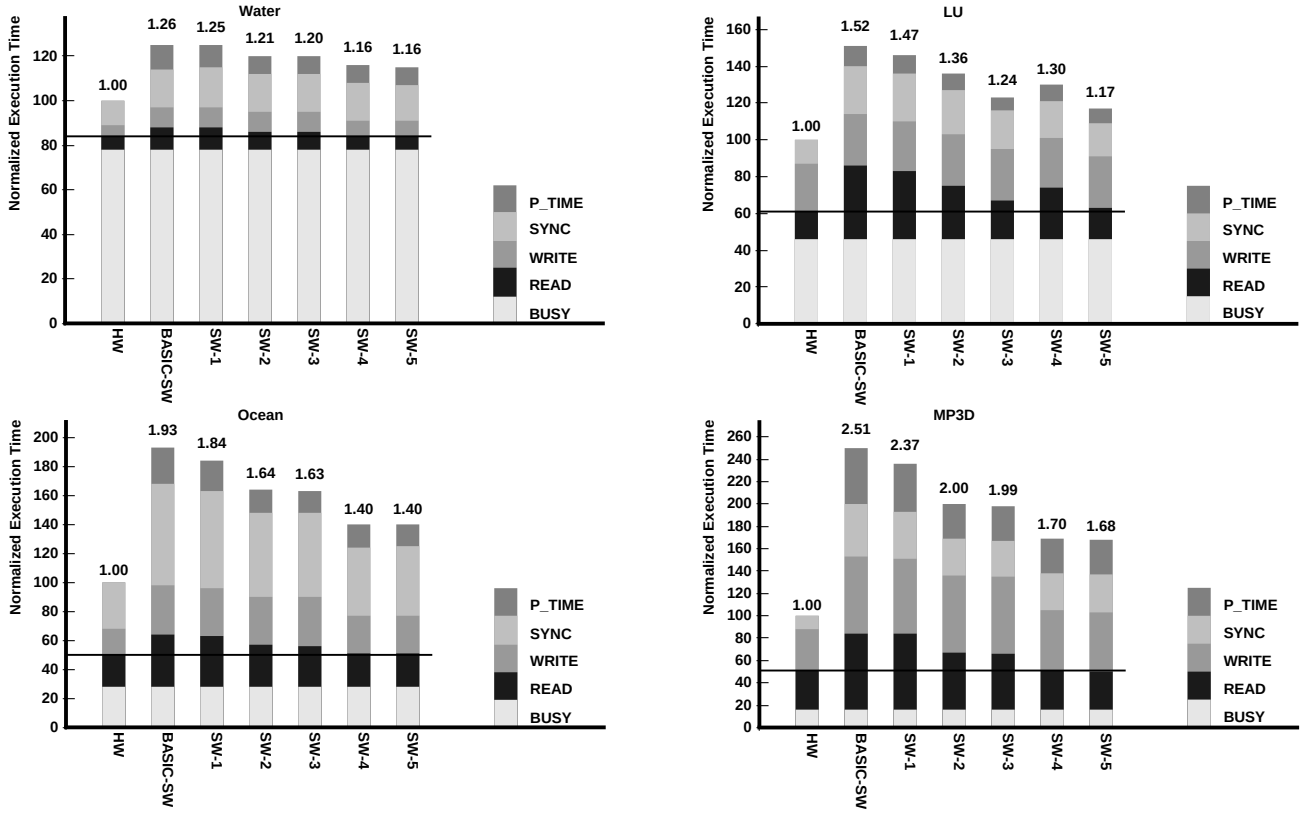

Figure 3:Execution times for the software-only protocols relative to the hardware-only protocol.

When we allow the node controller to forward read and write requests to dirty blocks directly to remote without interrupting the compute processor, as in SW-2, we would expect to see a dramatic reduction of the read stall time and the P_TIME evolving from the handling of dirty blocks. As expected, these components go down under SW-2 and result in an overall reduction of the execution time by between 4% (Water) and 20% (MP3D) as compared to BASIC-SW. We have seen that the fractions of all read misses that go to dirty blocks are as high as 87%, 85%, and 92% in Water, Ocean, and MP3D, respectively, and these applications demonstrate the highest gains of this strategy. Although cold misses dominate in LU (see Table 3) and only 15% of the misses go to dirty blocks, the execution time is 11% shorter than in BASIC-SW. This is because there are on average about 50% fewer interrupts pending in the processor interrupt buffer when a new message arrives.

In SW-3, the node controller returns data to local on read miss requests to clean blocks before interrupting the processor and is effective for applications with misses to clean blocks such as cold misses. As expected, since LU is the only application where the majority of misses are clean, this strategy is most effective for LU. The execution time for LU under SW-3 is 15% lower than under BASIC-SW. This is mainly an effect of a reduced read stall time, which is 45% lower under SW-3 than under BASIC-SW.

By letting the node controller return a dirty block to local before interrupting the processor as in SW-4, we further optimize the handling of dirty misses and ownership requests to dirty blocks and we can see that the read stall times are further reduced as compared to SW-2 for all applications. As expected, the effect is lowest for LU since, again, LU has a very low coherence miss rate. For the other three applications, the read stall time is reduced by 25%, 22%, and 31% for Water, Ocean, and MP3D, respectively, when we go from SW-2 to SW-4. The execution times are between 4% (Water) and 15% (MP3D) lower under SW-4 than under SW-2.

When we combine all strategies covered so far, which is done in SW-5, we can see additional gains for the software-only directory protocol. SW-5 has only 16% longer execution time than HW for the Water application meaning that it has 86% of HW's performance. Even for MP3D, which already for HW has a very poor processor utilization [14], we find that the software-only directory protocol has only 68% longer execution time. This result is encouraging, since we consider MP3D as a 'worst-case' application. We also observe that the read stall times under SW-5 is only between 2% (Ocean), 4% (Water), and 16% (LU) longer than under HW. LU has a relatively higher read stall time which we have traced to a high number of read miss retries.

In summary, we have seen that one can afford to let the compute processor update the directory given that the block data is transferred as soon as possible. In SW-3 and SW-4, the processor updates the directory in parallel with the reply sent to local, thus minimizing the miss penalty seen by local. Finally, SW-3 optimizes the handling of clean misses, while SW-4 optimizes the handling of dirty misses. Both strategies are essential to achieve good performance for both types of misses. They are combined in SW-5. Henceforth, we only consider SW-5 and refer to it as SW.

5.2 Limitations of Software-Only Protocols

While SW results in virtually the same read stall times as HW, the performance differences stem from the write and synchronization stall times and the time each processor handles coherence actions from other nodes through software handler executions. While we

focus on the write stall time in Section 5.3, we will in this section focus on the software latency and synchronization stall time.

Software handler execution overhead (P_TIME in Figure 3) shows up when a processor is interrupted and it is executing application code. By contrast, if the processor is stalled as a result of a cache miss, an ownership acquisition, or a synchronization operation, the software handler execution could be completely or partly overlapped by these stall time components. Thus, it becomes interesting to see to what extent software handling can be overlapped by the stall time components. In Table 4 we show this data for SW.

Surprisingly, only 18% or less of the software handler execution time is overlapped by the stall time components. To understand why this overlap is so small, Table 4 also summarizes the fraction of interrupts that occur when a processor is busy. Across the applications, we see that the processor is busy in at least 44% of the cases when an interrupt occurs. In Water, for example, the processor is busy in 72% of the cases which compares well with the processor utilization in Water which is 67%. By contrast, the processor utilization for MP3D is only 10%, but the processor is busy when 44% of the interrupts occur. We have found that this discrepancy stems from the fact that interrupts are clustered in MP3D; a new interrupt occurs within 50 pclocks after the previous one in 23% of the cases, within 100 pclocks in 53% of the cases, and finally, within 200 pclocks in 98% of the cases. Clearly, since the software latency is between 60 and 70 pclocks for SW, the processor does not have much time to do useful work between the interrupts. Overall, most of the software latency cannot be overlapped by the other stall time components because the processor is often busy when an interrupt occurs.

Table 4: Software handler execution statistics under SW.

	Application			
	Water	**LU**	**Ocean**	**MP3D**
Overlapped fraction of software handler execution	12%	6%	13%	18%
Fraction of interrupts that arrive when the processor is busy	72%	57%	47%	44%
Ratio: Highest/Lowest number of interrupts to any processor	1.84	2.26	4.87	10.19

Another important limitation to analyze stems from synchronization overhead. A striking observation extracted from Figure 3 is that the synchronization time under SW (i.e. SW-5) is much higher than under HW. This effect is especially pronounced in Ocean and in MP3D, where the synchronization stall times are more than twice as high for SW. An explanation for this difference would be that software handler execution causes load imbalance because the interrupts are not uniformly distributed across the processor nodes. To test this intuition, we recorded the highest and lowest numbers of interrupts coming to any processor for each application. The ratios of these numbers are shown in Table 4 for SW.

For Water and LU, which have the highest processor utilizations, the ratios are approximately 2, i.e., the processor that receives most interrupts, receives twice as many interrupts as the processor that is interrupted the fewest number of times. By contrast, Ocean and MP3D, the two applications that behave worst under SW, have a much higher ratio of the highest and the lowest number of interrupts; the ratios are around 5 and 10 in Ocean and in MP3D, respectively. This explains why these applications have a considerably higher synchronization overhead under SW than under HW. Ideally, if the coherence interrupts would be uniformly distributed among the processor nodes, we would expect the synchronization overhead to be the same as under HW. One reason that this does not happen is that we have not done any attempt to distribute data to avoid hot spots; our simulations assume that pages are allocated in a round-robin fashion.

5.3 Effects of Relaxed Consistency Models

The last component of the application execution time that differs between SW and HW is the write stall time. While we have seen that it is possible to achieve a nearly identical read stall time for SW and HW, the software handler execution time will appear on the critical memory access path for ownership acquisitions. The processor in the home node must examine the directory before sending invalidations, and as a result, the software handler execution time can not be removed from the write stall time seen by local. By contrast, under relaxed memory consistency models the write latency can be overlapped by application execution as long as synchronizations are not encountered. Therefore, this optimization is presumably more important for software-only protocols than for hardware-only protocols. However, since latency-tolerating techniques such as relaxed memory consistency models make the processors execute faster, we would expect that the rate of interrupts increases. This can potentially make the read and synchronization stall time components longer, which can offset the gain of the write stall time reduction.

To study the execution time effects under release consistency, we present in Figure 4 the resulting execution times for Ocean under release consistency (RC) for both HW and SW, normalized in each case to the execution times under sequential consistency (SC). In the release consistent systems, we include the full power of the lockup-free *SLCs* and their buffers. While we only present results for Ocean, the results for the other applications are similar. By analyzing the results for SW first, we see that the write latency is completely hidden and that the other components are virtually unaffected. However, the read stall time is slightly higher which stems from contention in the network rather than on secondary effects due to a higher interrupt rate. By comparing the execution time reduction of release consistency for HW and SW we see that it is virtually the same; the execution time is 18% shorter under RC than under SC for HW as well as SW.

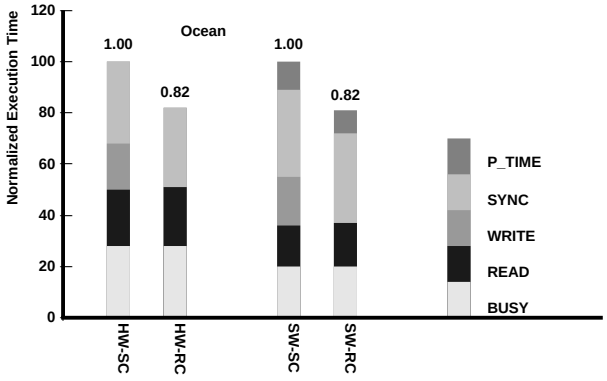

Figure 4: Execution time of HW and SW under sequential consistency (SC) and release consistency (RC) in Ocean. The execution time is normalized for each system to the execution time under sequential consistency.

5.4 Effects of Finite Caches

Until now we have assumed infinite second-level caches in our simulations. However, since finite caches introduce a higher miss rate as a result of replacements, it is important to study how the handling of replacements and misses affects the performance of the software-only protocols. We do so by presenting numbers derived from simulations using 16 Kbyte *SLCs*. In Table 3, we show the total read miss rates for 16 Kbyte caches. By comparing these numbers with the numbers for infinite *SLCs*, we can see that the miss rates have gone up significantly.

Upon replacement of a clean block, the cache can either notify the directory of the replacement or it could ignore to do so. In the latter case, the cache must be able to acknowledge invalidation messages even if its block copy does not exist. In a hardware-only implementation, this could result in a somewhat longer write stall time because of a larger number of invalidations, whereas in a software-only protocol one could benefit from fewer software handler invocations. We have simulated both alternatives and found, although the differences are small, that it is preferable not to notify the directory of replacements and take the cost of a higher number of invalidation messages. In Figure 5, we present the normalized execution times under HW and SW for the replacement strategy without directory notification.

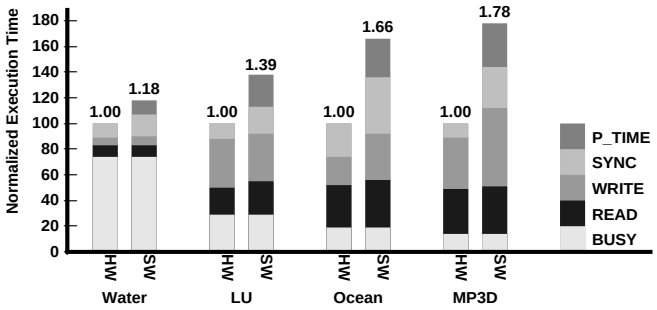

Figure 5: Execution time effects of finite second-level caches (16 Kbyte).

As expected, we see in Figure 5 that SW suffers more from the increased miss rate than does HW. By comparing the fraction of the execution time that stems from software handler overhead (P_TIME) in these simulations and the simulations assuming infinite *SLCs* in Figure 3, we can clearly see that P_TIME has gone up because of the higher miss rate; Water has 18% longer execution time under SW than under HW with finite caches, as compared to 16% longer execution time when we have infinite caches (see Figure 3). For the other three applications the difference is larger. Interestingly, the read stall time is virtually the same for HW and SW. This shows that our strategies efficiently handle read misses.

5.5 Handler Execution Time and Processor Speed

Until now, we have assumed that the software handler execution time is 50 pclocks (excluding directory accesses and message transmissions). However, it is important to see how the effectiveness of the strategies are affected by variations in this assumption. In Figure 6 we show the resulting execution times for Water and MP3D; the best and the worst performing applications. SW-50, SW-100, and SW-200 correspond to SW with a handler execution time of 50, 100, and 200 pclocks, respectively, again excluding directory access and message transmissions.

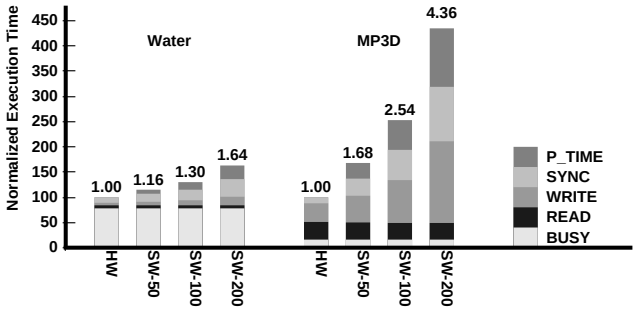

Figure 6: Effects of various protocol execution times.

As expected, we observe in Figure 6 an increased overhead for SW as the protocol execution time increases. Interestingly, the read stall times are virtually unaffected. This suggests that our strategies are successful in maintaining the same read stall time for SW as for HW. However, in LU we have seen an increased read stall time because of a high number of read-miss retries owing to that blocks are now locked a longer time. We also see that the write stall times and the protocol execution overheads are increased. This can be explained by the observations in Sections 5.1 and 5.2 that showed limited opportunities for overlap.

The next variation is motivated by the steadily increasing speed gap between commodity microprocessors and memory technology. We have simulated HW and SW assuming both a processor running at 100 MHz and a processor running at 300 MHz. All other latency times in the memory system are kept constant. In Figure 7, we show the resulting execution times of Water and MP3D under HW and SW when 100 MHz and 300 MHz processors are used.

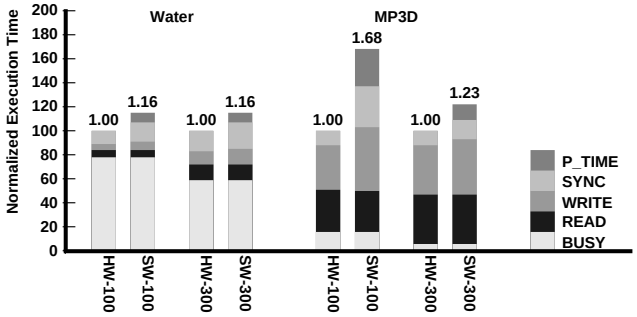

Figure 7: Execution times of SW relative to HW for 100 MHz and 300 MHz processors.

In Figure 7, we show the execution time of SW relative to HW for each processor speed. First, when comparing SW-100 with SW-300 we see that the stall time components in MP3D constitute a larger fraction of the overall execution time in the 300 MHz systems. As a result, P_TIME becomes a relatively smaller component in SW and the overhead of the software-only protocol becomes less. We also see that the speed gap between HW and SW is decreased when we consider faster processors; in the 100 MHz systems the speed gap is 68% whereas it is only 23% in the 300 MHz systems for MP3D. The same trends have been seen for the other applications.

6 Conclusions

The design complexity of hardware-based directory protocols has motivated us to study protocols in which the memory-protocol engine is migrated to software execution on the compute proces-

sor. In this paper we have focused on one such class, called software-only protocols, in which the directory is located in main memory and maintained by software handlers. All directory actions cause an interrupt on the compute processor which potentially can result in a significant protocol execution overhead. On a read miss, software handlers are invoked on the compute processor in the node where the block is allocated. Therefore, the latency of the read miss includes the time to execute the software handler. We have proposed several strategies to remove the latency of the software handler from the latency of a miss by executing the handler in parallel with the inter-node protocol transactions so as to overlap the software latency by network latency.

Based on architectural simulations using four parallel programs, we have evaluated six software-only protocol variations by comparing their performances with the performance of a hardware-only protocol. We have found that the software handler latency can be effectively removed from the critical memory access path for read misses, which resulted in a performance of the best strategy in the range 60%-86% of the hardware-only protocol. We have also identified the hardware functionality needed to support this strategy and it includes direct access to the data and state of each memory block. Software-only protocols also rely on processors supporting a fast mechanism to switch from execution of application code to software handler execution. Whereas we found that the software latency could only partially be removed from the critical path of ownership acquisitions, relaxed memory consistency models were found to be as effective for software-only protocols as for hardware-only protocols to eliminate the write stall time.

Two factors prevented further optimizations: overhead in servicing coherence actions for other processors and load imbalance due to nonuniform distribution of interrupts across nodes. Since we found that the processor is busy executing application code in a majority of the cases when an interrupt occurs, software handler execution will only partially be overlapped by the processor stall time components. Second, synchronization overhead is usually higher in a software-only protocol owing to the fact that some compute processors handle significantly more coherence actions than others. These factors dominated the performance difference between the software-only protocol and the hardware-only protocols and limited further gains. Finally, as indicated by our simulations, the performance difference between software-only and hardware-only protocols diminishes as we consider faster processors suggesting that software-only protocols will be more important in the future.

Overall, our study shows that if block data transfer is supported efficiently in hardware, one can afford to let the compute processor update the directory at a slower pace. Although this study gives some evidence that such an approach is promising, detailed implementation studies are needed to compare the complexity of alternatives.

Acknowledgments

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, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B-H. Lim, G. Maa, and D. Nussbaum, “The MIT Alewife machine: A large-scale distributed-memory multiprocessor”, in: M. Dubois and S.S. Thakkar, eds., Scalable Shared Memory Multiprocessors (Kluwer Academic Publishers, Boston, MA 1990) 240-261.

[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, J. Kubiatowicz, and A. Agarwal, “LimitLESS Directories: A Scalable Cache Coherence Scheme”, In *Proceedings of ASPLOS-IV*, pages 224-234, April 1991.

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

[7] A.L. Cox and R.J. Fowler, “Adaptive Cache Coherency for Detecting Migratory Shared Data”, In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98-108, May 1993.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15-26, May 1990.

[9] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, “Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor”, In *Proceedings of ASPLOS-VI*, pages 38-50, October, 1994.

[10] 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 ASPLOS-VI*, pages 274-285, October, 1994.

[11] D. S. Henry and C. F. Joerg, “A Tightly-Coupled Processor-Network Interface”, In *Proceedings of ASPLOS-V*, pages 111-122, October, 1992.

[12] 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.

[13] J. Kubiatowicz, D. Chaiken, and A. Agarwal, “Closing the Window of Vulnerability in Multiphase Memory Transactions”, In *Proceedings of ASPLOS-V*, pages 274-284, October 1992.

[14] 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.

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

[16] 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.

[17] P. Stenström, M. Brorsson, and L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing”, In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109-118, May 1993.

[18] 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 International Symposium on Computer Architecture*, pages 256-266, May 1992.