Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Systems Architecture 50 (2004) 537–561
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

A comparative evaluation of hardware-only and software-only directory protocols in shared-memory multiprocessors ☆

Håkan Grahn [a,*], Per Stenström [b]

[a] *Department of Software Engineering and Computer Science, Blekinge Institute of Technology, P.O. Box 520, SE-372 25 Ronneby, Sweden*
[b] *Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden*

Received 1 October 2001; received in revised form 5 February 2002; accepted 18 August 2003
Available online 3 February 2004

Abstract

The hardware complexity of hardware-only directory protocols in shared-memory multiprocessors has motivated many researchers to emulate directory management by software handlers executed on the compute processors, called *software-only directory protocols*.

In this paper, we evaluate the performance and design trade-offs between these two approaches in the same architectural simulation framework driven by eight applications from the SPLASH-2 suite. Our evaluation reveals some common case operations that can be supported by simple hardware mechanisms and can make the performance of software-only directory protocols competitive with that of hardware-only protocols. These mechanisms aim at either reducing the software handler latency or hiding it by overlapping it with the message latencies associated with inter-node memory transactions. Further, we evaluate the effects of cache block sizes between 16 and 256 bytes as well as two different page placement policies. Overall, we find that a software-only directory protocol enhanced with these mechanisms can reach between 63% and 97% of the baseline hardware-only protocol performance at a lower design complexity.

Keywords: Shared-memory multiprocessors; Cache coherence; Hardware-only directory protocols; Software-only directory protocols; Performance evaluation

1. Introduction

Private caches in conjunction with a directory-based cache coherence protocol have gained a lot of popularity in many shared-memory multiprocessor designs [1,21–23]. The memory-protocol engine that implements the directory-based protocol however contributes significantly to the total

☆ This paper is an extended version of "Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors" that was presented at the 22nd Int'l Symp. on Computer Architecture in June 1995.

* Corresponding author. Tel.: +46-457-38-58-04; fax: +46-457-271-25.
E-mail addresses: hakan.grahn@bth.se (H. Grahn), pers@ce.chalmers.se (P. Stenström).
URLs: http://www.ipd.bth.se/~hgr/, http://www.ce.chalmers.se/~pers/.

1383-7621/$ - see front matter
doi:10.1016/j.sysarc.2003.08.014

logic overhead and engineering effort of each processing node as demonstrated by the DASH design [22]. Furthermore, since the protocol is hard-wired, flexibility in its design and optimization is not provided.

Several efforts have addressed these concerns by migrating the entire protocol-engine functionality, or parts of it, to software handlers executed on a separate processor. In the Stanford FLASH [15] and the Wisconsin Typhoon [29] 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 design of a special-purpose processor apparently is costly.

A more radical approach has been taken in other research projects [1,17,28,31,32] where all or parts of the memory-protocol engine functionality is emulated by software handlers on the compute processor. Chaiken and Agarwal [7] evaluated a spectrum of such software-extended protocols with respect to performance and cost ranging from *software-only directory 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 invoked first when all the five hardware pointers are exhausted. While the software-only directory protocol design point is extremely attractive from a hardware complexity viewpoint, they found that it performs significantly worse than its hardware-only directory protocol counterpart. We will in this paper present a number of techniques that increase the performance of software-only directory protocols, making them more competitive with hardware-only protocols.

This paper focuses on the performance and implementation trade-offs between hardware-only and software-only directory protocols in the same architectural framework. The major difference between their performances is due to the impact of the software handler latency on the execution time. However, our evaluation reveals three common case operations that can be supported by simple hardware mechanisms and can make the performance of software-only directory protocols competitive with hardware-only protocols with a reduced engineering effort. The first technique aims at removing the software handler latency from the critical memory access path for cache misses to data in the local memory. The second technique removes software handler latency from the critical memory access path of misses to data in remote memories and caches by reducing or overlapping the software handler latency with the latency of remote miss transactions. Finally, we also consider to cut the software handler latency itself by enabling caching of directory information in the compute processor caches.

Besides the identification of the common case operations to boost performance of software-only directory protocols, an important contribution of this paper is to determine performance trade-offs between the hardware-only and the software-only directory protocol design points. This is done in the same architectural simulation framework consisting of a cache-coherent NUMA machine with a full-map write-invalidate directory protocol that is driven by eight applications from the SPLASH-2 suite [35]. We find that the performance of the software-only directory protocol using the mechanisms for efficient miss handling of local and remote application data misses is competitive with that of the hardware-only protocol. In addition, our data suggest that directory caching can improve overall application performance significantly; it only marginally interferes with caching of application data while significantly reducing the software handler latency. In total, the software-only directory protocol performance is between 63% and 97% of the hardware-only directory protocol performance for all applications.

Another important contribution of this paper is to identify the design trade-offs between hardware-only and software-only directory protocols. While the enhanced software-only protocols we propose require some minor modifications to the memory controllers and network interfaces in each node, most of the protocol complexity is migrated to software. We have also implemented a prototype of the network interface supporting our suggested hardware mechanisms.

As for the rest of the paper, we present in the next section our architectural framework and the baseline hardware-only and software-only directory protocols we simulate. Section 3 then presents a number of simple enhancement techniques to boost the performance of software-only directory protocols. In Sections 4 and 5, we present our experimental methodology and discuss the results from our simulations, respectively. Then, in Section 6 we present and discuss our prototype implementation of the network interface. Finally, in Section 7 we put our work in context of related work before reaching our conclusions in Section 8.

2. Hardware-only and software-only directory protocols

The performance and design trade-offs between hardware-only and software-only directory protocols are evaluated in the same architectural framework which is introduced in Section 2.1. Sections 2.2 and 2.3 then present the node design requirements of the baseline hardware-only and software-only directory protocols, respectively. Our discussion centers on the functionality and implementation aspects of the protocol engines associated with the memory modules; the protocol engines associated with the caches are identical for the two types of protocols.

2.1. Architectural simulation framework and coherence protocol

The architectural framework consists of a cache-coherent NUMA machine (CC-NUMA) with a node organization according to Fig. 1. Each node consists of a processor with a two-level cache hierarchy and associated write buffers with a similar organization as many contemporary microprocessors.

This paper centers on the mechanisms needed to support a directory protocol to maintain consistency across the second-level caches in all the nodes. Let us therefore briefly review the directory protocol we assume which is similar to the Censier and Feautrier protocol [5].

A presence flag vector with one flag per node is associated with each memory block in the node in which the corresponding page is mapped, called the *home node*. In addition, a few state bits encode whether the memory copy is clean, dirty or in transition between any of these states. Read-miss and ownership requests to transient memory blocks are forced to be retried as in the DASH [22]. A second-level cache (*L2*) miss request is sent to the home of the memory block. If the memory copy is clean, the miss is immediately serviced by *home* which records the identity of the requesting node (denoted *local*) in the directory and returns a copy of the block. In contrast, if the block is dirty,

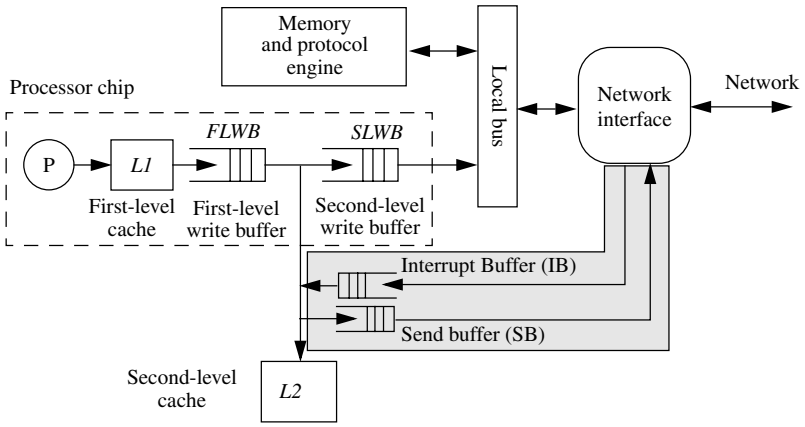

Fig. 1. Organization of the processor nodes. The shaded areas only apply to the software-only directory protocol that is discussed in Section 2.3.

home forwards the request to the node keeping the only copy (denoted *remote*); *remote* writes the block back to *home*; and finally, *home* returns the block to *local* resulting in as many as four node-to-node 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 *L2* 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. We follow the recommendation in [7,26] and assume for all protocols we consider that the network interface in Fig. 1 has the capability to collect acknowledgments from remote nodes and notifies the memory-protocol engine when the last acknowledgment to a pending write request has arrived. Finally, one design option is whether or not to notify *home* when a block is replaced from a cache. If not, it may cause a larger number of invalidations and longer write stall times for the hardware-only directory protocol but fewer handler invocations for software-only directory protocols. Simulations revealed that the performance differences for these alternatives were small for hardware-only protocols. However, for software-only directory protocols the latter showed some advantage which is why we will assume that in the following.

2.2. Simulated baseline hardware-only directory protocol

A central part of the hardware-only implementation of the protocol in the previous section is the memory-protocol engine in Fig. 1 that interfaces to the network and to the local cache via the local bus.

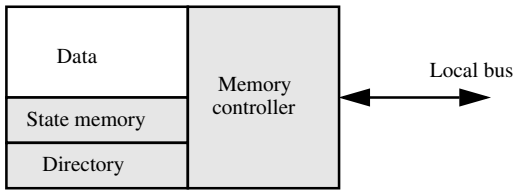

Fig. 2. The memory module organization for a hardware-only directory protocol.

The memory-protocol engine conceptually consists of three parts, which are shaded in Fig. 2: the directory, the state memory, and the memory controller. The directory uses N bits, where N is the number of nodes in the system, and the state memory uses three bits for each block.

The memory controller implements the coherence protocol actions. This controller typically processes an incoming request by carrying out the following tasks: it decodes the message; it performs a directory and state-memory lookup, and possibly a state modification according to the coherence protocol; 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 [22]. This has motivated many researchers to consider software-only directory protocols whose baseline we present next.

2.3. Simulated baseline software-only directory protocol

The functionality of the memory-protocol engine is emulated by software handlers on the compute processor in our baseline software-only directory protocol; all requests arriving at the home directory will interrupt the processor on that node which executes the corresponding handler.

Since all coherence requests are taken care of by software handlers on the compute processor, the directory and state bits for each memory block assumed by the hardware-only protocol can now

be allocated in the memory. More importantly, the hard-wired memory controller for the hardware-only directory protocol is replaced by software handlers.

To enable the emulation of the protocol actions requires some functionality and organizational modifications as compared to a hardware-only directory protocol. First, the processor must support a low-level interrupt mechanism that rapidly can switch between program execution and protocol execution. Second, an Interrupt Buffer (IB, shaded in Fig. 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 [34] are assumed to rapidly select which coherence routine the processor shall execute. IB is interfaced to the second-level cache bus as proposed in [16], 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, shaded in Fig. 1) 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. We assume that the buffers use memory-mapped addresses instead of register-mapped as in [16] to adjust to mainstream processor designs.

Coherence interrupts, also called *high-availability interrupts* [19], need to be precise as pointed out in [19] 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* [19]. 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. *Associative locking* [19], 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 [22], some request- response dependencies are solved using separate network 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* [20]. 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, while the memory-protocol engine now can be managed in software, software-only directory protocols require that processors support a fast mechanism to switch between normal execution and software handler execution, and also lockup-free first and second-level caches to interrupt a pending load instruction so as to avoid deadlock. Let's next build intuition into the relative performance of the hardware-only and software-only directory protocol baselines in this section.

3. Enhancement techniques for software-only directory protocols

The difference in performance between the hardware-only and software-only directory protocols in the previous section stems from the latency of the software handlers that are invoked at the *home* node for each read miss and ownership request. The increase in read stall and write stall times is one source of performance overhead because the software handler latency appears on the critical path of cache miss and ownership handling. Another performance overhead is incurred at the compute processor in *home* when handling requests initiated by other nodes. We have identified three simple enhancements to common case operations of the baseline software-only directory protocol. These enhancements, which either seek at removing the software handler latency from the critical access path or reducing it, are as follows:

- Eliminating software handler latency from the critical path of misses read to the local memory. This technique is described in Section 3.1.
- Eliminating software handler latency from the critical path of read misses to remote memory or caches. This technique is described in Section 3.2.
- Reducing software handler latency by enabling caching of directory information. This technique is described in Section 3.3.

3.1. Optimization of read miss requests to the local node

In order to avoid software handler invocation on cache misses to data in the local node, we augment the software-only directory protocol implementation with a separate state memory, as in the hardware-only implementation, as shown in Fig. 3. A memory controller is added that returns a block copy to the local processor as long as the block is clean, i.e., no software handler invocation is needed. 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 if it came from another node and puts the miss request in the IB which 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 Fig. 3 is much simpler than the controller in Fig. 2 because it only has to retrieve the state of the memory block instead of implementing the entire protocol.

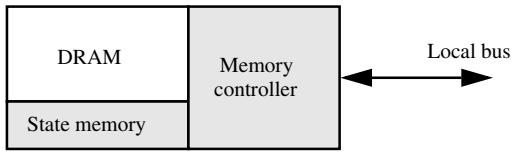

Fig. 3. The memory module organization in a software-only directory protocol with support for efficient handling of read misses to local memory.

Since a read miss to a clean block located in the same node does not interrupt the compute processor, the directory—which is still managed by software handlers—is not updated. In Alewife [1,6,7], a special local-bit is used to indicate whether the local cache has a copy of the block or not [7]. 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 (to other nodes) on the bus in order to maintain coherence in the local cache. Note that this would be difficult in a node organization such as in SGI Origin that uses a node controller instead of a bus [21]. 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

Let us consider techniques that remove the software handler latency from the access path of misses to data in other nodes by first developing a simple performance model for this case.

Fig. 4 shows the latencies involved in a read-miss transaction to a memory block that is dirty in a remote node in a scenario where *local*, *home*, and *remote* all are different nodes. A read miss to a dirty block incurs the latency of four network hops

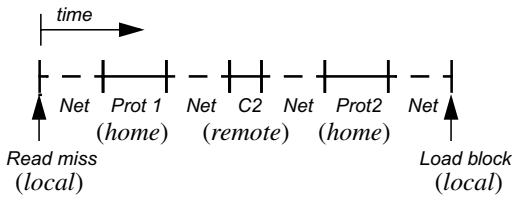

Fig. 4. Timing of a read miss to a dirty memory block.

(*Net*) plus the latency of two software handler executions at the home node (*Prot1* and *Prot2*) plus the time to retrieve the block from the remote cache (*C2*). In contrast, *Prot1* and *Prot2* in a hardware-only directory protocol are smaller in comparison to *Net*. 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 following techniques aim at doing so. They require some minor modifications to the network interface of Fig. 1.

3.2.1. Performing state-memory lookup in hardware, SW-1

We have already exposed the state memory bits of a memory block to the memory controller to avoid software handler invocations on misses from the local compute processor. It is natural to do the state memory lookup for miss requests from other nodes in hardware which thus frees the software handlers from this common case operation. 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 network interface, *Prot1* and *Prot2* in Fig. 4 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, i.e., the processor is not interrupted in that case. As we will see, this strategy enables other enhancement techniques.

To implement this, the network interface must have access to the state of a memory block. To check whether a block is in a transient state, the network interface 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 network interface already routes and sends messages, it can send retries with virtually no extra functionality. This enhancement technique combined with the enhancement of misses to local data is called ***SW-1***.

3.2.2. Reducing the software handler latency of dirty misses, SW-2

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 network interface can have dramatic effects on the processor stall time components. One strategy is to let the network interface forward read and write requests to dirty memory blocks directly to *remote* without interrupting the processor which potentially removes *Prot1* in Fig. 4 from the stall time incurred by dirty misses.

To support this strategy, the network interface 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 network interface 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 enhancement as ***SW-2***.

3.2.3. Hiding software handler latency of clean misses, SW-3

Misses to clean blocks that reside in another node than *local* incur two network traversals plus handler execution time ($Net + Prot1 + Net$ in Fig. 4). Once we let the network interface perform state lookup, it is possible to remove *Prot1* from the critical memory access path by letting the network interface 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 misses to clean as well as 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 enhancement apart from those of SW-2, is the ability for the network interface to read the block from memory. A new network interface 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 network interface 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***.

3.2.4. Hiding software latency of dirty misses, SW-4

Whereas SW-2 and SW-3 can remove *Prot1* in Fig. 4 from the critical memory access path for misses to dirty and clean blocks, the next strategy aims at removing *Prot2* from read miss and ownership transactions to dirty blocks. This strategy is built on top of SW-2 and lets the network interface 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 network interface 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***.

3.2.5. Combining all optimization strategies for remote misses, SW-5

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 network interface 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 directory 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 network interface 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. Note however that the controller associated with the network interface is significantly simpler in that it only does request forwarding based on the state bits and does not have to manipulate the directory itself. Table 1 summarizes its functionality and the expected performance effects of each optimization.

Table 1
Optimization strategies for software-only directory protocols

Strategy	Functionality in the network interface	Expected performance enhancement
SW-1	Memory state lookup by network interface as well as processor	Optimizes local misses and can slightly reduce *Prot1* and *Prot2* for remote misses
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

3.3. Reducing software handler latency by caching directory information

Since the latency of the software handler is charged on the compute processor in *home*, 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. Conversely, caching of directory data can interfere with application code and data caching. The net performance effect is thus not clear.

When a directory entry is cached, another block—possibly belonging to the application—can be evicted from the cache. If the evicted block is needed later, this results in a replacement miss. This miss may incur a long latency if it is handled by a remote node. Thus, a higher access penalty may be encountered by the application program as a result of directory caching. Assume that the compute processor encounters a miss rate of M_A and M_S for the application and the software handlers, respectively, and that the corresponding miss penalties are MP_A and MP_S respectively. The average miss penalty for the compute processor is then:

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

Because 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 Section 5 we will quantitatively evaluate whether it pays off to cache directory data.

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 Sections 2.3 and 3. The simulation models are built on top of the CacheMire Test Bench [4], 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 according to Fig. 1, and in Table 2 we have collected the architectural parameters. Memory pages are allocated using a static round-robin policy to simplify the analysis. We will also evaluate a first-touch (FT) policy, i.e., a page is allocated to the node whose processor first accesses it. Instruction references and references to private data are assumed to always hit in the *L1* cache, i.e., they do not incur any memory system latencies. The default memory consistency model in this study is sequential consistency and the processor is stalled on each shared data access until it has completed. Acquires and releases are supported by a queue-based lock mechanism, similar to the one implemented in the DASH [22]. In the software-only directory protocols this mechanism is implemented by software handlers.

The only difference between the network interfaces for the hardware-only and the software-only directory 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 directory protocols and we do not address them in this study; simulations assume infinite buffers.

Table 3 shows the time it takes to satisfy a read miss request from different levels in the memory hierarchy in the hardware-only implementation assuming no contention. In our simulations, however, a request usually takes longer time as a result of contention. Contention is correctly modeled in all parts of the processor nodes. Clearly, in a software-only directory protocol,

Table 2
Architectural parameters

Parameter	Value and comments
Block size	64 bytes
First-level (*L1*) and second-level (*L2*) cache sizes	8 Kbyte *L1* cache (write-through, direct-mapped) + 256 Kbyte *L2* cache (full inclusion, direct-mapped, lockup-free)
Write-buffer size	16 Entries
Page size	4 Kbyte
Processor speed	1 GHz, single-issue (1 pclock = 1 ns)
L1 access time	1 ns = 1 pclock
L2 access time (for a block)	8 ns = 8 pclocks (SRAM, interleaved)
Memory access time (for a block)	60 ns = 60 pclocks (DRAM, interleaved)
Interrupt and send buffer access times	5 ns = 5 pclocks
Bus speed (128-bits wide)	200 MHz, 5 ns arbitration + 5 ns transfer
Network interface speed	200 MHz, 800 Mbytes/s into and out of each node
Network latency (infinite bandwidth)	75 ns = 75 pclocks (approximately the latency in a 200 MHz mesh network with 32-bit flits)
Software handler execution time	100 pclocks + additional time as described in the text

Table 3
Average read miss latency times for a hardware-only implementation assuming a conflict-free system

Latency numbers for read requests	1 pclock = 1 ns (1 GHz)	Comment
Fill from *L1* cache	1 pclock	
Fill from *L2* cache	8 pclocks	
Fill from local memory	88 pclocks	*L1* + *L2* look-up, 2 local bus transactions, 1 memory access
Fill from *Home* (clean, 2-hop)	288 pclocks	*L1* + *L2* look-up, 4 bus transactions, 1 memory access, 2 network 'hops'
Fill from *Remote* (dirty, 4-hop)	598 pclocks	*L1* + *L2* look-up, 6 bus transactions, 2 memory access, 4 network 'hops', 1 remote *L2* look-up

software handler invocations also prolong the request latencies.

A critical timing assumption is how many cycles we charge for the software handlers. 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 (in processor clock cycles) from the optimized Sparcle processor described in [2] and use them as a base for how fast a software handler can be dispatched.

We assume 100 pclocks as the default protocol execution time which does not include the time to access the directory and state information. Rather, these 100 pclocks include the following actions: interrupt handling (4 pclocks); reading the message from the IB (10 pclocks); dispatch of the corresponding software handler (4 pclocks); administration of the software directory (78 pclocks); and restart of the application program (4 pclocks). On top of this we charge 130 pclocks to read or write the global state of a memory block; 2 or 160 pclocks for directory access if the directory is cached or not, respectively, and the same to read or write a memory block; and finally, 10 pclocks to send a message depending on the type of coherence action. For example, for a read miss to a clean block, 65 pclocks (read the state) plus 80 pclocks (read the block) plus 10 pclocks (send the reply) plus 80 pclocks (update the directory) are added to the basic 100 pclocks, resulting in 335 pclocks to service a read miss to a clean block. Note that the processor in *home* is interrupted and application

execution in *home* is prolonged with the handler execution time.

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 looks as follows (in C-syntax):

$$\text{Address}_{\text{directory}} = 0\text{x}70000000 | ((\text{Address}_{\text{data}}/\text{block\_size}) * 4)$$

The state and directory information of a block are not cached by default in the simulations. We will experimentally study the performance impact of directory caching in Section 5.2. For SW-1 to SW-5 we also assume that a dedicated state memory is supported.

4.2. Benchmark programs

In order to understand the relative performance of hardware-only and the variations of the software-only directory protocols, we use 8 scientific and engineering applications written in C using the ANL macros and compiled by `gcc` (version 2.7.2) with optimization level-O2. All applications are from the SPLASH-2 suite [35].

A short description of the applications together with the data set sizes we use are shown in Table 4. 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. The applications chosen have different characteristics. In Cholesky, FFT, and Radix cold misses dominate, while coherence misses dominate in the other applications. Barnes has an unpredictable communication pattern, while the other applications have a more regular and predictable communication pattern. By using a 256 Kbyte second-level cache the primary working set for all applications fits in the cache [35].

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 study the effects of directory caching, in Section 5.3, we analyze what limits further improvements, and finally in Section 5.4, we evaluate how different cache block sizes impact the performance of software-only directory protocols.

5.1. Efficiency of software-only directory protocol strategies

In this section, we evaluate the relative effectiveness of the different strategies for software-only directory protocols proposed in Section 3.2 by comparing their performances with the baseline hardware-only and software-only directory protocols in Section 2. We use the round-robin page

Table 4
The parallel programs together with the data set sizes we use in our simulations

Application	Description	Data set size/input data
Barnes	Barnes–Hut hierarchical N-body simulation	16 384 particles
Ocean	Simulate eddy currents in an ocean basin	258×258 grid
Water-Nsquared	Molecular dynamics simulation, $O(N^2)$ algorithm	512 molecules, 3 time steps
Water-Spatial	Molecular dynamics simulation, $O(N)$ algorithm	512 molecules, 3 time steps
Cholesky	Blocked sparse Cholesky factorization	tk29.O
FFT	1-D SQRT (n) six-step fast Fourier transform	64K points
LU	Blocked LU-decomposition of a dense matrix	512×512 matrix, 16×16 blocks
Radix	Integer radix sort	256K keys, radix 1024

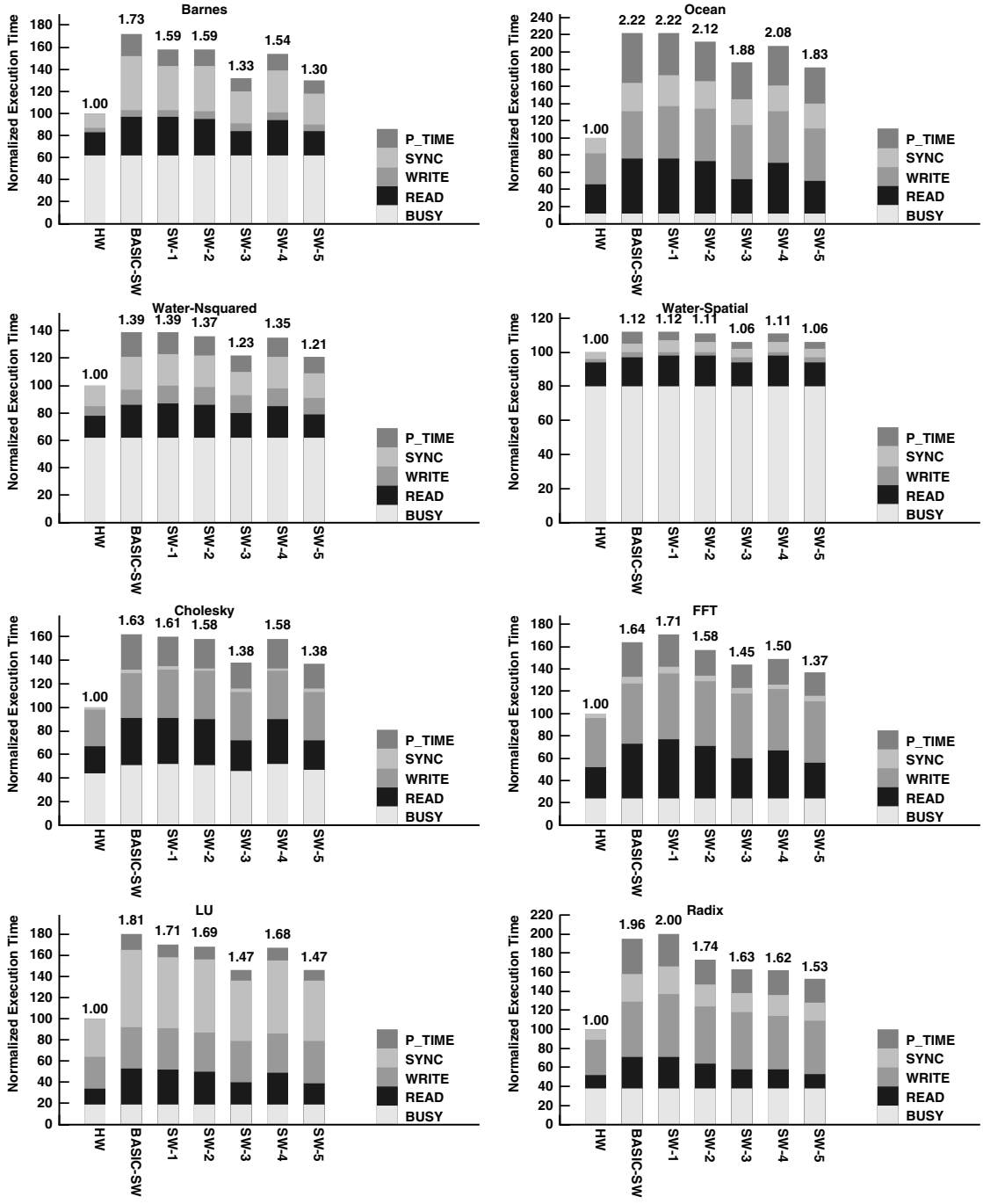

Fig. 5. Normalized execution times for the software-only directory protocol optimizations relative to the hardware-only directory protocol.

allocation strategy in this section in order to interpret the results more easily. In Fig. 5, we show the relative execution times of the different optimization strategies. All execution times are nor-

malized 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) directory protocols in Sections 2.2 and 2.3, respectively. They are followed by the strategies listed in Table 1 in Section 3.2. 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, i.e. stall times for acquiring or releasing a lock; and finally, for software-only directory protocols, the overhead in servicing coherence actions by software-handler execution (P_TIME).

Let us first compare the baseline protocols. As expected, BASIC-SW does significantly worse than HW; the execution times are between 12% (Water-Spatial) and 122% (Ocean) 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 5 that shows the cold, coherence, and replacement read-miss rates. Since the applications differ in the number of misses, we also see a big difference in the relative performance; while e.g., Water-Spatial has few misses (0.19%) and requires few software handler invocations, protocol handling dominates in Ocean which has a high miss rate (1.48%) 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.

Starting with SW-1 (as defined in Table 1), in which state lookup is performed by the network interface, we would expect P_TIME to be reduced. This is true for all applications as can be seen in Fig. 5. However, for most applications, the write stall time increases slightly. When the network interface does state lookup, the latency seen by a write request arriving at a node until the reply leaves the node actually increases. In BASIC-SW, the processor fetches both the state and the directory for a block at the same time. In contrast, in SW-1 the state is first fetched by the network interface, and then the directory data is fetched by the processor resulting in one additional interaction with the memory controller and two extra bus transfers. In total, the resulting execution time under SW-1 is shorter than or the same as under BASIC-SW for all applications except FFT and Radix. For these two applications, the write stall time increases more than for the other applications since they already have longer write stall times than the other applications.

When we allow the network interface 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 reduction of the read stall time and the P_TIME evolving from the handling of dirty blocks. As expected, these components go down to various degrees under SW-2 for most applications. This results in an overall reduction of the execution

Table 5
Application miss rates of a software-only directory protocol, with 256 Kbyte *L2* caches and round-robin page allocation

Application	Application miss rates					Handler miss rates
	Cold (%)	Coherence (%)	Replacement (%)	Directory conflict (%)	Total (%)	
Barnes	0.10	1.14	0.07	0.06	1.37	7.18
Ocean	0.02	0.69	0.70	0.07	1.48	3.68
Water-Nsquared	0.04	0.29	0.08	0.02	0.43	18.68
Water-Spatial	0.03	0.16	0.00	0.00	0.19	1.64
Cholesky	0.21	0.12	0.13	0.02	0.48	6.70
FFT	1.19	0.29	0.05	0.03	1.56	3.11
LU	0.07	0.16	0.17	0.00	0.40	0.42
Radix	0.45	0.35	0.26	0.07	1.13	3.04

time by up to 13% (Radix) compared to BASIC-SW. The reason that we see relatively small benefits from SW-2 is that the fractions of all read misses that go to dirty blocks are low; below 10% for all applications except FFT (27%) and Radix (29%). In a previous study [12], we found that about 90% of the read misses were to dirty blocks for three of four applications from the older SPLASH suite, and thus we saw much larger gains from SW-2 in that study.

In SW-3, the network interface 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, e.g., cold misses. In the SPLASH-2 applications, much data is updated by one processor and read by many which results in many misses to clean blocks. Thus, SW-3 shows dramatic reductions of mainly the read stall time for all applications. The read stall time is reduced by between 40% (LU) and 17% (Water-Spatial) under SW-3 as compared to BASIC-SW. In total, the execution times under SW-3 is between 5% (Water-Spatial) and 23% (Barnes) lower 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. We can see that the read stall times are only slightly reduced as compared to SW-2 for most applications. SW-4 is not so effective for the same reason as SW-2; only a small fraction of the read misses is to dirty blocks. SW-4 has the largest effect on Radix where the read stall time and execution time drop by 21% and 7% respectively, when we go from SW-2 to SW-4. In [12] we observed significantly larger gains from SW-4 because misses to dirty blocks were more common in the applications used in that study.

When we combine all strategies covered so far, which we do in SW-5, we can see additional gains for the software-only directory protocol. SW-5 has only 6% longer execution time than HW for the Water-Spatial application meaning that it has 94% of HW's performance. For Ocean, which already for HW has a very poor processor utilization, we find that the software-only directory protocol has 83% longer execution time, i.e., it reaches 55% of HW's performance. We have also observed that the read stall times under SW-5 is at most 11% longer than under HW for all applications but LU (37% longer).

In summary, we have seen that one can afford to let the compute processor update the directory given that parallelism is exploited between message transfer and protocol actions. 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*. 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.

5.2. Effects of caching the directory information

In this section we evaluate the effects of caching the directory in the processor. In Fig. 6, we show the resulting execution times for a software-only directory protocol without (SW-5) and with (SW-5 + Dir) directory caching normalized to the execution time of a hardware-only protocol (HW). In the figure results are shown for both round-robin (RR) and first-touch (FT) page allocation.

By comparing the execution times of RR/SW-5 and RR/SW-5 + Dir in Fig. 6, we find that they are reduced by between 3% (Water-Spatial) and 20% (Ocean). The reduction stems from a lower handler execution time that reduces the write and synchronization stall times as can be seen in Fig. 6. 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 RR/SW-5 + Dir is only 3–46% longer than under RR/HW. In [12], where the older SPLASH suite was used, we varied the *L2* cache size from 4 Kbyte to infinitely sized *L2* caches and found the results consistent for all applications with the results for the 256 Kbyte *L2*, i.e., *caching the directory results in shorter execution times*.

To understand the results above, recall the simple miss penalty formula from Section 3.3:

$$T_{\text{miss}} = M_{\text{A}} * MP_{\text{A}} + M_{\text{S}} * MP_{\text{S}}$$

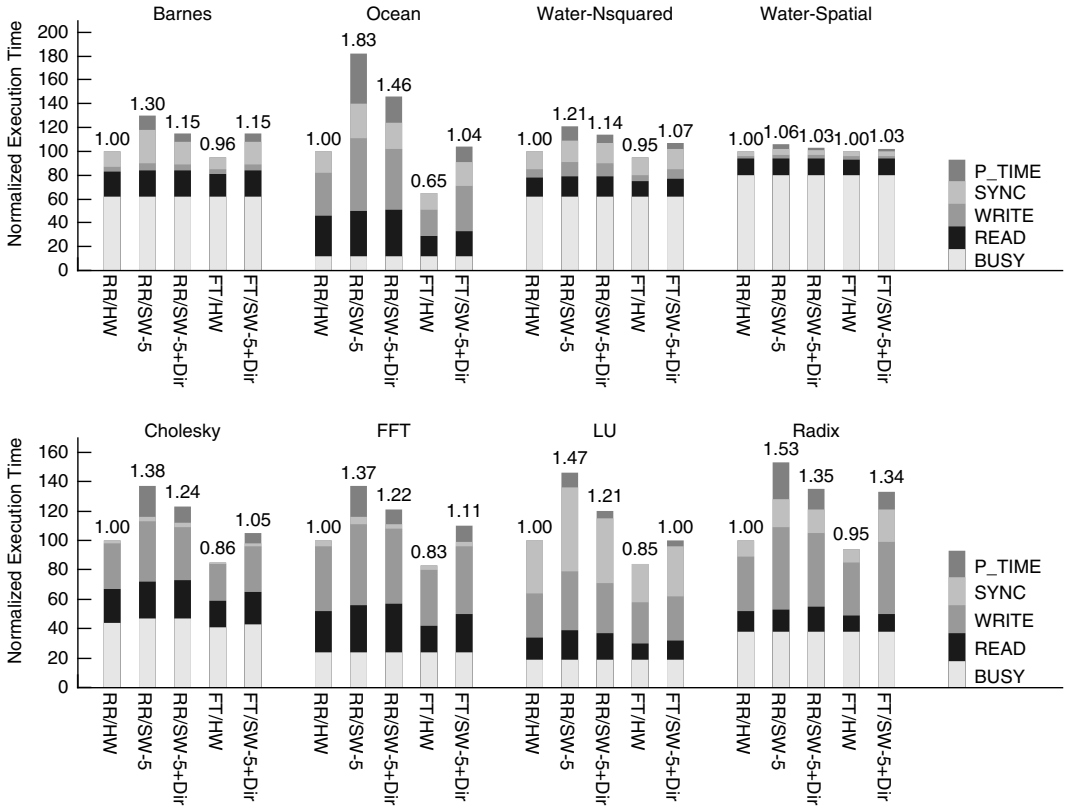

Fig. 6. Normalized execution times for the applications with a hardware-only protocol (HW) and a software-only directory protocol without (SW-5) and with (SW-5 + Dir) directory caching, using two different page allocation policies: round-robin (RR) and first-touch (FT).

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 applications and the software handlers. The resulting miss rates are summarized in Table 5.

The directory-conflict column in the table is the miss rate an 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 Barnes only 0.06% as compared to 1.37%. Therefore, having a separate small cache for directory data will probably not be worth implementing considering the relatively small potential performance improvement. Further, we see in Table 5 that the miss rates for the software handlers are only between 0.42% (LU) and 7.18% (Barnes) for all applications but one. 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. For Water-Nsquared, the handler miss rate is 18.68%. This indicates that there is less locality in the directory accesses for that application.

In Fig. 6 we also present results using a better page allocation strategy, first-touch (FT). The motivation is that by allocating pages close to the processors that use them, the read miss latency

should decrease. Further, in the software-only directory protocol we can take advantage of the optimization of local misses as described in Section 3.1. We can observe that for some applications, e.g., Barnes and Water-Spatial, the page allocation policy does not impact the performance. For other applications, e.g., Ocean, Cholesky, and LU, we see significant performance improvements by using a better page allocation policy. Comparing the performances of FT/HW and FT/SW-5 + Dir we find that the software-only directory protocol has between 3% (Water-Spatial) and 60% (Ocean) longer execution times than the hardware-only protocol.

In summary, when both the local and remote miss optimizations are applied and the directory information is cached, as in SW-5 + Dir, the execution times are only between 3% and 60% 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 low, only between 0.42% and 7.18%, for seven of eight applications which indicates a high locality in the directory accesses. Henceforth, we only consider FT/SW-5 + Dir and refer to it as ***SW***.

5.3. Limitations of software-only directory 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, starting with the handler execution overhead.

Software handler execution overhead (P_TIME in Fig. 5) shows up when a processor is interrupted while executing application code. In 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 handler execution can be overlapped by the stall time components. In Table 6 we show this data for SW.

Simulation results show that at most 46%, and sometimes as low as only 9%, of the software handler execution time is overlapped by the stall time components. To understand why this overlap is so small, Table 6 also summarizes the fraction of interrupts that occur when a processor is busy as well as the processor utilization. Across the applications, we see that the processor is busy between 11% and 73% of the cases when an interrupt occurs. In Barnes, for example, the processor is busy in 61% of the cases which compares well with the processor utilization in Barnes which is 57%.

In contrast, the processor utilization for FFT is 25%, but the processor is busy only when 11% of the interrupts occur. Such a large discrepancy is also present for Water-Nsquared and Cholesky. We have found that the discrepancy stems from the fact that interrupts are clustered in these

Table 6
Software handler execution statistics under the SW directory protocol

Application	Overlapped fraction of software handler execution (%)	Fraction of interrupts that arrive when the processor is busy vs. processor utilization	Ratio: highest/lowest number of interrupts to any processor
Barnes	17.29	60.60% vs. 57%	2.89
Ocean	45.68	17.08% vs. 13%	1.35
Water-Nsquared	36.19	35.42% vs. 61%	2.24
Water-Spatial	8.66	73.25% vs. 79%	555.06
Cholesky	45.81	23.93% vs. 44%	1.61
FFT	26.27	11.04% vs. 25%	1.15
LU	37.08	25.19% vs. 20%	2.49
Radix	35.95	25.93% vs. 32%	3.70

applications; e.g., for Cholesky a new interrupt occurs within 200 pclocks after the previous one in 25% of the cases, within 500 pclocks in 68% of the cases, and finally, within 1000 pclocks in 88% of the cases. Clearly, since the average software latency is between 120 and 140 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.

Another important limitation to analyze stems from synchronization overhead. An observation extracted from Fig. 6 is that the synchronization time under SW (i.e., FT/SW-5 + Dir) is higher than under HW. This effect is especially pronounced in Barnes, Water-Nsquared, LU, and Radix. 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 6 for SW.

For Barnes the ratio is 2.89, i.e., the processor that receives most interrupts, receives almost 3 times as many interrupts as the processor that is interrupted the fewest number of times. For Water-Nsquared, LU, and Radix the corresponding ratios are 2.24, 2.49, and 3.70, respectively. This explains why Barnes, Water-Nsquared, LU, and Radix have a considerably higher synchronization overhead under SW than under HW. In contrast, for all other applications except one (Water-Spatial) the ratio is well below two which indicates a good balance of the number of interrupts between the processors. For Water-Spatial, however, the processor that receives most interrupts handles more than 500 times as many interrupts as the processor with the fewest number of interrupts. This is explained by the fact that some nodes have only one page allocated to them, thus they only need to handle very few interrupts. Since the total number of interrupts for Water-Spatial is low, the performance is not significantly affected by this load imbalance. In [12], we found interrupt ratios as high as 10 for MP3D, an application in the older SPLASH suite. Ideally, if the coherence interrupts would be uniformly distributed among the processor nodes, we would expect the synchronization overhead to be virtually the same as under HW.

Finally, the software handler execution time will appear on the critical memory access path for ownership acquisitions since the processor in *home* 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*. An approach to address the problem is to use a relaxed memory consistency model [11], where the write latency can be overlapped by application execution as long as synchronizations are not encountered. In [13], we evaluate the performance effects of relaxed models as well as other latency tolerating techniques.

5.4. Effects of block size variations

An important design parameter is the size of a memory block. Larger block sizes have a potential to reduce the cache miss rate, i.e., the read stall time, and at the same time reduce the protocol execution overhead. Too large a block size may introduce false sharing [10], which may increase both the number of invalidations for each write and the cache miss rate. In order to understand what the trade-offs are in the context of hardware-only and software-only protocols, we simulated block sizes of 16, 32, 64, 128, and 256 bytes. The resulting execution times are shown in Fig. 7, where HW-X and SW-X denote the hardware-only and the software-only directory protocols, respectively, and X the block size in bytes. The execution times are normalized to the execution time of HW-64, i.e., a hardware-only protocol with 64-byte blocks.

Considering the execution times for the hardware-only directory protocol we note that for four of the applications (Barnes, Ocean, Cholesky, and FFT) we observe a constant decrease of the execution time as the block size increases from 16 bytes up to 256 bytes. However, remember that we do not model contention in the interconnection network, and thus, the execution time estimations are optimistic for larger block sizes. For the other

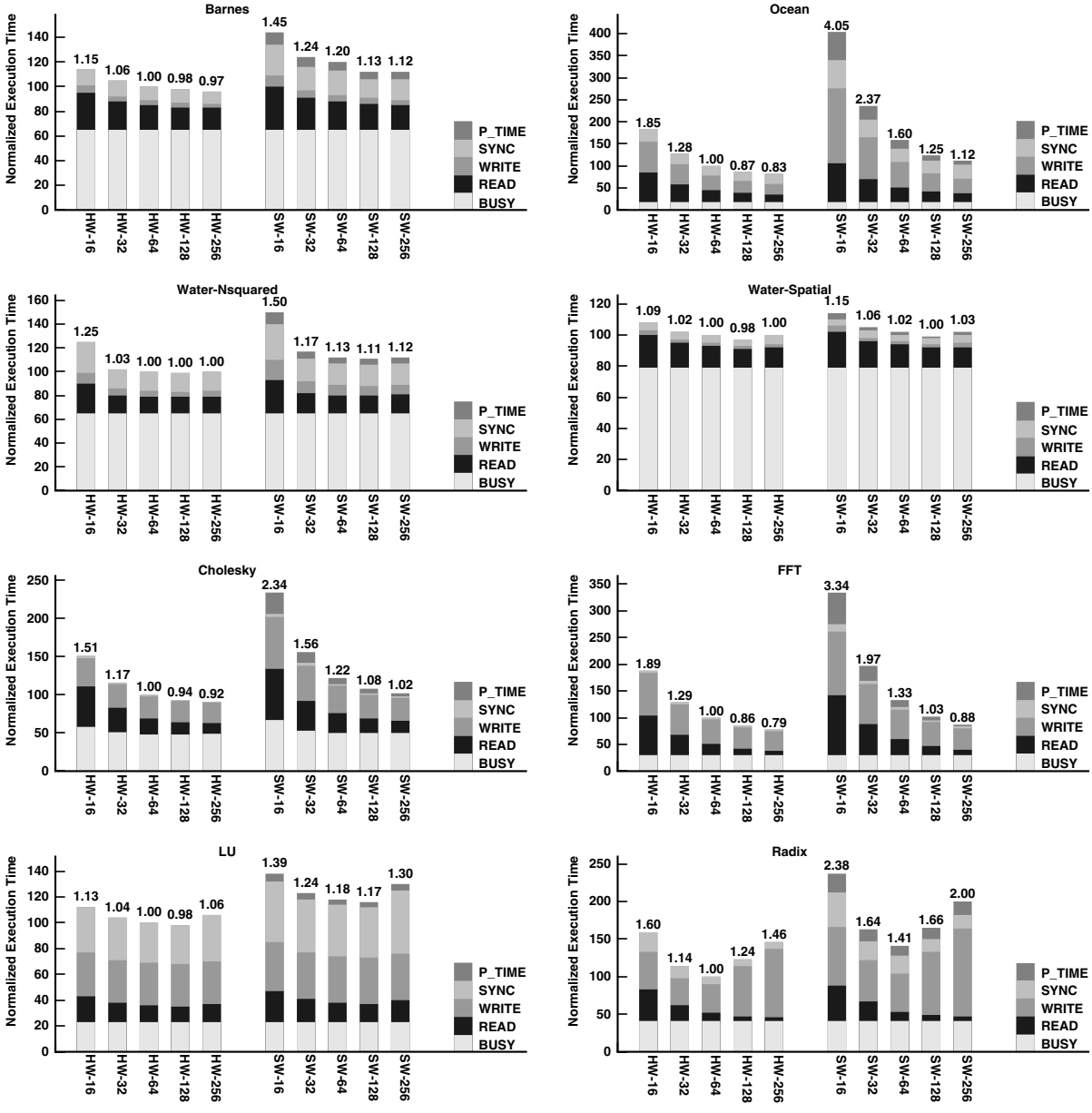

Fig. 7. Normalized execution times of the applications for 16, 32, 64, 128, and 256 byte blocks.

applications the best block size, as indicated by our results, seems to be 64 (Radix) or 128 bytes (Water-Nsquared, Water-Spatial, and LU). Larger block sizes increase the execution time as a result of false sharing [10] for those applications.

By looking at the execution times for the software-only directory protocol, similar effects as for the hardware-only protocol can be observed but they are more pronounced. For four of the applications (Barnes, Ocean, Cholesky, and FFT) the execution time decreases more rapidly than for the

hardware-only directory protocol as the block size increases from 16 to 256 bytes. For Water-Nsquared, Water-Spatial, and LU, the execution time decreases up to a block size of 128 bytes, and then it increases slightly again. Finally, for Radix, we see a steep decrease in execution time for increasing block sizes up to 64 bytes, and then the execution time increases rapidly as a result of false sharing. As expected, the results in Fig. 7 show that the protocol execution overhead decreases with larger block sizes for all applications until false sharing occurs. Then, both the write stall time and the protocol execution overhead increases as false sharing increases.

In contrast to the execution time variations between different block sizes for the hardware-only directory protocol, the execution time variations are much more dramatic for the software-only directory protocol. In other words, the performance of the software-only directory protocol is more sensitive to the block size choice than the hardware-only directory protocol performance is. The ratios between the execution times for the worst block size choice and for the best block size choice are between 1.11 (Water-Spatial) and 2.39 (FFT) for the hardware-only directory protocol. The corresponding ratios for the software-only directory protocol are 1.15 (Water-Spatial) and 3.80 (FFT), which indicates the higher sensitivity to the block size choice.

For all applications, the execution time ratios between software-only and hardware-only directory protocols generally decrease as the block size increases, i.e., the execution time decreases faster for the software-only than for the hardware-only directory protocol for increasing block sizes. Larger block sizes result in fewer coherence interrupts, which then decreases the performance difference between hardware-only and software-only directory protocols.

6. Implementation of the optimization strategy

In this section we discuss some issues concerning the hardware complexity. We have made a design of a network interface, as well as a prototype VHDL implementation using Mentor Graphics' System Design Station. The prototype has been synthesized to a 0.8 μm full custom technology and simulated, but not manufactured.

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 Fig. 3). 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. Fig. 8 shows a high-level overview of the network interface which is one example implementation that we use in this study. The extra logic required for the software-only approach as compared to a hardware only approach is shaded in Fig. 8. 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

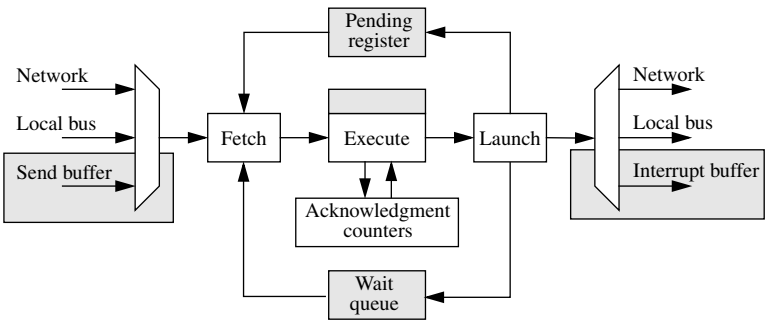

Fig. 8. A logic overview of the network interface.

operations according to the contents of a message. The other boxes are referred to in the text when necessary.

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 Fig. 8). 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 a 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 a 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 Fig. 8). 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.

7. 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 [14,15] and the Wisconsin Typhoon [29]. 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. In Typhoon-0 and Typhoon-1 [30], the network interface and the protocol processor are decoupled, i.e., a commodity processor in the processing node is used for protocol processing. 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 [1,6], the Cooperative Shared Memory [17,36], and the StarT-NG [8] 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 [7], 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. In contrast, our results indicate that also software-only directory protocols can be competitive with hardware-only protocols.

Wood et al. evaluate mechanisms for Cooperative Shared Memory in [36] and compare different hardware-only and software-extended protocols. The software-extended protocol that is most similar to our work is called Dir_1SW+. It can in hardware keep track of one shared cached copy of the block. Further, it also supports request forwarding to dirty remote blocks and then passing the block directly to the requesting cache as in our SW-4 strategy. However, when the number of shared copies exceeds 1, a software handler is executed in order to *broadcast* invalidations upon a write. Our interpretation is that the Dir_1SW+ protocol does not keep track of which nodes have a copy of a memory block if more than one copy exists. This approach can have a devastating effect on performance in large systems. In contrast, our approach is to always let the directory maintain exact information about which nodes have a copy of a block. Finally, in our study we have used larger data sets for the benchmarks and provide a much more detailed analysis of which strategies are most important to support in hardware.

In [27], Qin and Baer study the performance of a software-based coherence protocol in the context of SMP clusters. Their approach relies on a separate protocol processor in the cluster node. They find that the performance of the software scheme, extended with some extra forwarding logic similar

to our suggestions and software hints, is comparative to the performance of a pure hardware scheme.

An interesting approach is found in [28], where a software-based shared memory system called DSZOOM-WF is presented. They implement an all-software protocol on a system where each node is an SMP cluster. In DSZOOM-VF all coherence protocol actions are run on the requesting processor, which otherwise stalls waiting for data, and thus removes all protocol software overhead. In the paper, the software-based approach demonstrates performance comparable to a hardware-based shared memory implementation.

The StarT-NG [8] 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 is 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. The succeeding StarT-Voyager [3] project suggests to add an extra service processor in the network interface, similar to the Wisconsin Typhoon proposal.

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 several processors. 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. In 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 [32], 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.

In [25], Moga et al. propose and evaluate a software-controlled COMA (SC-COMA) architecture. They rely on a special hardware unit (Access Checking Device) to detect misses in the attraction memory, i.e., the local memory, in each node. Upon an attraction memory miss, the processor is interrupted and a software handler is executed and a remote request is issued. In contrast, in our approach the local processor is not interrupted for a miss in the local node. Instead, a remote request is issued directly from hardware as in a hardware-only implementation. As for the home memory node, both our approach and SC-COMA support caching of directory information, but do not allow tag/state information to be cached since it is accessed by both software handlers and the hardware. In SC-COMA, there is no support for request forwarding in hardware for misses to dirty blocks in remote nodes, i.e., the main processor is interrupted for each request arriving to *home*. Finally, also the processor in *remote* is interrupted in SC-COMA when servicing a read miss to a dirty block. In total, servicing a dirty remote miss interrupts three processors in SC-COMA while only the processor in *home* is interrupted in our approach.

Shasta [31] is a software-only directory protocol targeted for a pure message-passing substrate and implements a relaxed consistency model, therefore aiming in a slightly different direction than our approach. As a pure software approach, Shasta also provides more flexibility than our approach. Shasta relies on rewriting the application to introduce code that does fine-grain access control for each load and store to shared data. This potentially increases the code size and execution time of the application. Further, Shasta uses a polling scheme in the *home* nodes instead of interrupts to detect when a message arrives. *Home* does polling for incoming requests when the protocol waits for a reply. In addition, the access control code is inserted at function calls or loop back edges, further increasing the code size and the execution time for the application. The polling overhead can be significant for applications with little degree of sharing which leads to many unnecessary polling actions. Finally, no comparison between Shasta and a hardware-only implementation with similar architecture has been done.

In Section 5.3, we found that a performance limitation of software-only directory protocols is that the processor often is busy executing application code when an interrupt occurs. This can be addressed by, e.g., using multiple processors in each node. Then, the possibility to find an idle processor that can handle the coherence interrupt increases [18]. Another way to address the problem is to use a processor supporting simultaneous multithreading [33], such as the Intel Xeon processor [24] or the Compaq Alpha 21464 [9]. Then, the software handler execution could be integrated in the processor's instruction stream without disturbing the execution of the application code.

8. 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 handlers executed on the compute processor. In this paper we have focused on one such class, called software-only directory 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.

In this paper we have studied three important performance issues for software-only directory protocols: read misses to local memory; read misses to remote nodes; and the effects of caching the directory when executing a software handler. We have also 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 eight parallel benchmarks from the SPLASH-2 suite [35], we have evaluated seven software-only directory protocol variations by comparing their performances with the performance of a hardware-only directory protocol. We have found that the software handler latency can be effectively removed from the critical memory access path for read misses. Further, 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. In total, the best software-only directory protocol resulted in a performance in the range 63–97% of the hardware-only directory protocol performance.

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. For example, 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. Further, software-only directory protocols also rely on processors supporting a fast mechanism to switch from execution of application code to software handler execution.

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 often busy executing application code 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 directory 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 and the hardware-only directory protocols and limited further gains.

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, more detailed implementation studies are needed to compare the complexity of alternatives.

Acknowledgements

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 Stanford University, as well as the anonymous reviewers for their valuable comments.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, D. Yeung, The MIT Alewife machine: Architecture and performance, in: Proc. 22nd Int'l Symp. Comp. Arch., June 1995, pp. 2–13.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, M. Parkin, Sparcle: An evolutionary processor design for large-scale multiprocessors, IEEE Micro. 13 (3) (1993) 48–61.

[3] B.S. Ang, D. Chiou, D.L. Rosenband, M. Ehrlich, L. Rudolph, Arvind, StarT-Voyager: A flexible platform for exploring scalable SMP issues, in: Proc. Supercomputing'98, November 1998.

[4] M. Brorsson, F. Dahlgren, H. Nilsson, P. Stenström, The CacheMire test bench—A flexible and effective approach for simulation of multiprocessors, in: Proc. 26th Ann. Simulation Symp., March 1993, pp. 41–49.

[5] L.M. Censier, P. Feautrier, A new solution to coherence problems in multicache systems, IEEE Trans. Comput. C-27 (12) (1978) 1112–1118.

[6] D. Chaiken, J. Kubiatowicz, A. Agarwal, LimitLESS directories: A scalable cache coherence scheme, in: Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), April 1991, pp. 224–234.

[7] D. Chaiken, A. Agarwal, Software-extended coherent shared memory: Performance and cost, in: Proc. 21st Int'l Symp. Computer Architecture, April 1994, pp. 314–324.

[8] D. Chiou, B.S. Ang, R. Greiner, Arvind, J.C. Hoe, M.J. Beckerle, J.E. Hicks, A. Boughton, StarT-NG: Delivering seamless parallel computing, in: Proc. EURO-PAR'95, Lecture Notes in Computer Science, No. 966, Springer-Verlag, Berlin, 1995, pp. 101–116.

[9] K. Diefendorff, Compaq chooses SMT for Alpha, Microprocessor Report 13 (16) (1999) 5–11.

[10] M. Dubois, J. Skeppstedt, P. Stenström, Essential misses and data traffic in coherence protocols, J. Parallel Distributed Comput. 29 (2) (1995) 108–125.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, in: Proc. 17th Int'l Symp. Computer Architecture, May 1990, pp. 15–26.

[12] H. Grahn, P. Stenström, Efficient strategies for software-only directory protocols in shared-memory multiprocessors, in: Proc. 22nd Int'l Symp. Computer Architecture, June 1995, pp. 38–47.

[13] H. Grahn, P. Stenström, Comparative evaluation of latency-tolerating and -reducing techniques for hardware-only and software-only directory protocols, J. Parallel Distributed Comput. 60 (7) (2000) 807–834.

[14] J. Heinlein, K. Gharachorloo, S. Dresser, A. Gupta, Integration of message passing and shared memory in the Stanford FLASH multiprocessor, in: Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 1994, pp. 38–50.

[15] 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, J. Hennessy, The performance impact of flexibility in the Stanford FLASH multiprocessor, in: Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 1994, pp. 274–285.

[16] D.S. Henry, C.F. Joerg, A tightly-coupled processor-network interface, in: Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), October, 1992, pp. 111–122.

[17] M.D. Hill, J.R. Larus, S.K. Reinhardt, D.A. Wood,, Cooperative shared memory: Software and hardware for scalable multiprocessors, ACM Trans. Comput. Syst. 11 (4) (1993) 300–318.

[18] M. Karlsson, P. Stenström, Performance evaluation of a cluster-based multiprocessor built from ATM switches and

bus-based multiprocessor servers, in: Proc. Second Int'l Symp. High Performance Computer Architecture (HPCA-2), February 1996, pp. 4–13.
[19] J. Kubiatowicz, D. Chaiken, A. Agarwal, Closing the window of vulnerability in multiphase memory transactions, in: Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), October 1992, pp. 274–284.
[20] J. Kubiatowicz, A. Agarwal, Anatomy of a message in the Alewife multiprocessor, in: Proc. 7th ACM Int'l Conf. Supercomputing, July 1993, pp. 195–206.
[21] J. Laudon, D. Lenoski, The SGI Origin: A CC-NUMA highly scalable server, in: Proc. 24th Ann. Int'l Symp. on Computer Architecture, June 1997, pp. 241–251.
[22] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, J. Hennessy, The DASH Prototype: Logic overhead and performance, IEEE Trans. Parallel Distributed Syst. 4 (1) (1993) 41–61.
[23] T. Lovett, R. Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, in: Proc. 23rd Int'l Symp. on Computer Architecture, June 1996, pp. 308–317.
[24] D. Marr et al., Hyper-threading technology architecture and microarchitecture, Intel Technol. J. 6 (1) (2002) 1–12, Available from <http://developer.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf>.
[25] A. Moga, A. Gefflaut, M. Dubois, Hardware versus software implementation of COMA, in: Proc. 26th Int'l Conf. on Parallel Processing, August 1997.
[26] J. Piscitello, A software Cache coherence protocol for Alewife, Master's thesis, Department of Electrical Engineering and Computer Science, MIT, May 1993.
[27] X. Qin, J.-L. Baer, Optimizing software Cache-coherent cluster architectures, in: Proc. Supercomputing'98, November 1998.
[28] Z. Radovic, E. Hagersten, Removing the overhead from software-based shared memory, in: Proc. Supercomputing'2001, November 2001.
[29] S.K. Reinhardt, J.R. Larus, D.A. Wood, Tempest and Typhoon: user-level shared-memory, in: Proc. 21st Int'l Symp. Computer Architecture, April 1994, pp. 325–336.
[30] S.K. Reinhardt, R.W. Pfile, D.A. Wood, Decoupled hardware support for distributed shared memory, in: Proc. 23rd Int'l Symp. on Computer Architecture, June 1996.
[31] D. Scales, K. Gharachorloo, C. Thekkath, Shasta: A low overhead, software-only approach for supporting fine-grain shared-memory, in: Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996, pp. 174–185.
[32] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, D.A. Wood, Fine-grain access control for distributed shared memory, in: Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 1994, pp. 297–306.
[33] D.M. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, in: Proc. 22nd Int'l Symp. Computer Architecture, June 1995, pp. 392–403.
[34] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, Active messages: a mechanism for integrated communication and computation, in: Proc. 19th Int'l Symp. Computer Architecture, May 1992, pp. 256–266.
[35] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: Proc. 22nd Int'l Symp. Computer Architecture, June 1995, pp. 24–36.
[36] D.A. Wood, S. Chandra, B. Falsafi, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, S.S. Mukherjee, S. Palacharla, S.K. Reinhardt, Mechanisms for cooperative shared memory, in: Proc. 20th Int'l Symp. Computer Architecture, May 1993, pp. 156–167.

Håkan Grahn is an Assistant Professor of Computer Engineering at Blekinge Institute of Technology. He received a M.Sc. degree in Computer Science and Engineering in 1990 and a Ph.D. degree in Computer Engineering in 1995, both from Lund University. His main interests are computer architecture, shared-memory multiprocessors, and performance evaluation. He has authored and co-authored thirty papers on these subjects and graduated one Ph.D. student. He received the Best Paper Award at the PARLE'94 (Parallel Architectures and Languages, Europe) conference. From January 1999 to June 2002 he was head of department for the Department of Software Engineering and Computer Science. Dr. Grahn is a member of the ACM, the IEEE, and the Computer Society. For more information, please refer to URL http://www.ipd.bth.se/~hgr/.

Per Stenström is a Professor of Computer Engineering and Vice-dean of the School of Computer Science and Engineering at Chalmers University of Technology, Sweden. His research interests are devoted to design principles for high-performance computer systems. He is an author of two textbooks and close to a hundred research publications. He has served on more than 30 program committees of major conferences in the computer architecture and parallel processing field and is an editor of IEEE Trans. on Computers and the Journal of Parallel and Distributed Computing. He was the General Chair of the ACM/IEEE 28th Int. Symposium on Computer Architecture. Dr. Stenström is a Senior member of the IEEE and a member of the ACM.