Journal of Parallel and Distributed Computing **60**, 807–834 (2000)
doi:10.1006/jpdc.1999.1606, available online at http://www.idealibrary.com on IDEAL®

Comparative Evaluation of Latency-Tolerating and -Reducing Techniques for Hardware-Only and Software-Only Directory Protocols[1]

Håkan Grahn[2]

Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Soft Center, S-372 25 Ronneby, Sweden
E-mail: Hakan.Grahn@ipd.hk-r.se

and

Per Stenström

Department of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden
E-mail: pers@ce.chalmers.se

Received November 10, 1997; accepted October 22, 1999

We study in this paper how effective latency-tolerating and -reducing techniques are at cutting the memory access times for shared-memory multiprocessors with directory cache protocols managed by hardware and software. A critical issue for the relative efficiency is how many protocol operations such techniques trigger. This paper presents a framework that makes it possible to reason about the expected relative efficiency of a latency-tolerating or -reducing technique by focusing on whether the technique increases, decreases, or does not change the number of protocol operations at the memory module. Since software-only directory protocols handle these operations in software they will perform relatively worse unless the technique reduces the number of protocol operations. Our experimental results from detailed architectural simulations driven by six applications from the SPLASH-2 parallel program suite confirm this expectation. We find that while prefetching performs relatively worse on software-only directory protocols due to useless prefetches, there are examples of protocol optimizations, e.g., optimizations for migratory data, that do relatively better on

[1] This paper is an extended version of "Relative Performance of Hardware- and Software-Only Directory Protocols under Latency Tolerating and Reducing Techniques" that was presented at the 11th International Parallel Processing Symposium in Geneva in April 1997.

[2] To whom correspondence should be addressed. Fax: +46-457-271 25.

0743-7315/00 $35.00

software-only directory protocols. Overall, this study shows that latency-tolerating techniques must be more carefully selected for software-centric than for hardware-centric implementations of distributed shared-memory systems.

Key Words: shared-memory multiprocessors; cache coherence; software-only directory protocols; prefetching; memory consistency models; migratory sharing; performance evaluation.

1. INTRODUCTION

Private caches and a directory-based cache coherence protocol implemented in hardware, also called a *hardware-only directory protocol*, constitute an important approach to achieving high performance in many recent distributed shared-memory multiprocessor designs [1, 27–29]. However, in order to reduce the hardware complexity and/or increase the flexibility, many researchers have considered migrating the coherence protocol, or parts of it, to software [1, 5, 7, 14, 20, 21, 34, 35]. Shared virtual memory (SVM) systems go even further by completely supporting the protocol mechanisms at the operating system or application level using the virtual memory system (see, e.g., [23]). We will refer to distributed shared-memory systems on which the directory protocol is implemented in software and run on a compute processor as *software-only directory protocols* [5].

In hardware-only as well as software-only directory protocols performance is often limited by processor stall times resulting from *memory access latencies*. To reduce processor stall times, and thus increase performance, several *latency-tolerating and -reducing techniques* have been proposed and evaluated in the context of hardware-only directory protocols [8, 10, 13, 19, 31, 36]. In addition to the processor stall times, the invocation of software handlers on the compute processor in software-only directory protocols can prolong the execution time. While the handler latency might end up on the memory access path and thus increase the memory access latency seen by the requesting processor, it is possible to reduce this latency component if message transmission is done in parallel with protocol processing [14]. A much more challenging problem is that protocol handling can result in a high compute processor occupancy which can degrade overall performance significantly [14] because of few opportunities to overlap protocol handling with other high-latency operations. While this problem can be reduced by increasing protocol handling throughput using clustering [24], it cannot be completely eliminated. The impact of latency-tolerating and -reducing techniques on compute processor occupancy is the major problem studied in this paper.

To illustrate the problem let us consider data prefetching, a latency-tolerating technique shown to be effective in hardware-only directory protocols [9, 31]. By using special prefetch requests, data is brought into the cache prior to its use, thus reducing the memory stall time. Because it is nearly impossible to implement a prefetch scheme that only issues prefetches that will eliminate misses, software-only directory protocols will suffer from a higher protocol handling occupancy which may offset the reduced memory stall time for eliminated misses. On the other hand, latency-tolerating or -reducing techniques that reduce the number of protocol

handler invocations are expected to be more effective on software-only than on hardware-only directory protocols.

In this paper we introduce a framework for reasoning about the relative efficiency of latency-tolerating and -reducing techniques on hardware-only and software-only directory protocols. The key observation is whether a technique is expected to *increase*, *decrease*, or *not affect* the protocol processing overhead in terms of number of protocol invocations. The relative performance between hardware-only and software-only directory protocols is expected to increase in the first and last case whereas it is expected to decrease in the second case. To demonstrate the usefulness of our framework, we apply it to three example techniques that represent these cases: *prefetching* [9, 10, 30]; *migratory optimization* [8, 16, 36], a technique that detects and eliminates protocol actions for migratory data blocks; and *release consistency* [13]. These techniques increase, reduce, and do not affect the protocol execution overhead, respectively.

Our experimental observations are based on an aggressive software-only and hardware-only directory protocol implementation of a CC-NUMA multiprocessor similar to DASH [28] that we simulate in detail. We drive our simulations with six applications from the SPLASH-2 parallel program suite [38] and find that prefetching, a technique that increases the overhead, and that typically improves the performance of hardware-only directory protocols, actually *degrades* the performance of software-only directory protocols in many cases. Further, we find that the migratory optimization technique, which reduces the overhead, is relatively more efficient on software-only than on hardware-only directory protocols. Finally, release consistency, which virtually does not affect overhead, successfully hides the write latency for both classes of systems across all applications. However, the relative performance difference between software-only and hardware-only directory protocols usually increases. A limitation of our study is that we only consider one design point with a quite aggressive software-only directory implementation. However, this means that the impact of a technique on the compute processor occupancy is expected to be much higher for less aggressive implementations. This emphasizes the importance of carefully selecting latency-tolerating and -reducing techniques for software-only directory protocols.

Section 2 first introduces the simulated architectural framework that allows us to concretely reason about the performance trade-offs followed by the assumptions that drive our experiments in Section 3. Then, in Sections 4–6 we focus on three example techniques and present our experimental findings. Finally, in Section 7 we discuss and generalize our findings before we conclude the paper in Section 8.

2. SIMULATED ARCHITECTURES AND THEIR PERFORMANCE DIFFERENCES

In Section 2.1 we first describe the hardware-only and software-only directory protocols we simulated. In Section 2.2 we then introduce a simple performance model that helps us to reason about the performance trade-offs between hardware-only and software-only directory protocols in general but using the architectural framework in Section 2.1 as a case study.

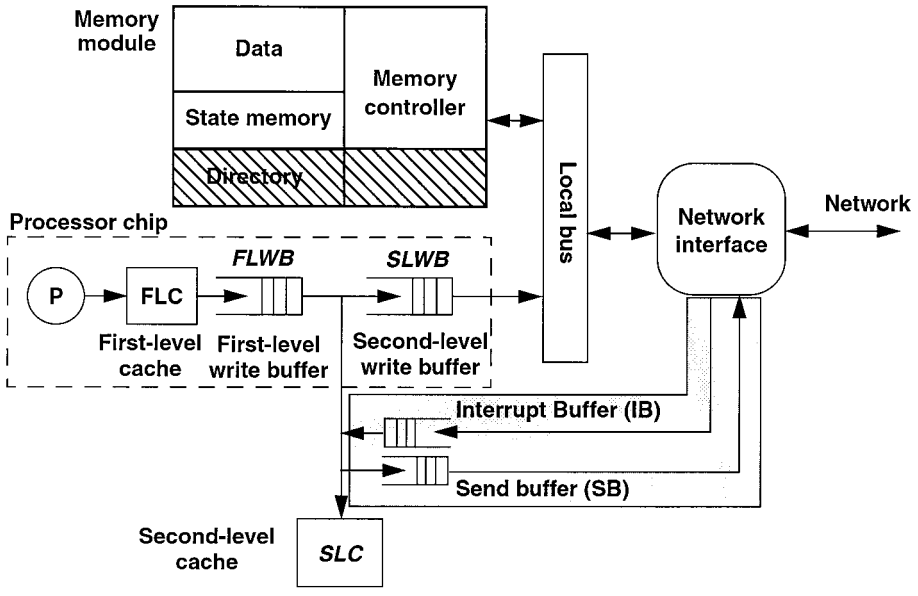

FIG. 1. The organization of a processor node. The striped area only applies to hardware-only directory protocols and the shaded area only applies to software-only directory protocols.

2.1. Simulated Hardware-Only and Software-Only Directory Protocol Architectures

The architectural framework is a sequentially consistent cache-coherent NUMA architecture similar to DASH [28], where a number of processor nodes are connected by a network. Each node consists of a processor with its cache hierarchy, a memory module, a local bus, and a network interface connecting the processor node to the network as shown in Fig. 1.

The hardware-only and the software-only directory systems both employ a write-invalidate protocol with a full-map directory [4]; i.e., for each memory block a presence flag vector is used to indicate which nodes have a copy of the block. A processor read operation that misses in the second-level cache (SLC) initiates a read miss request which is sent to the node where the memory block is mapped, denoted the *home* node. If the memory copy is clean, home responds with a block copy to the *local* requesting node and updates the directory. Otherwise, if another cache, denoted *remote*, has an exclusive and possibly modified copy, home issues a write-back request to remote; remote then updates home; and finally, home forwards a block copy to local, updates the directory, and the block ends up clean in home.[3]

A processor write to a nonexclusive block in the cache results in an ownership request sent to home. Home inspects the directory and sends explicit invalidations to the other caches with a block copy. Each cache receiving an invalidation responds with an acknowledgment to home. When home has collected all acknowledgments, it grants ownership to local and the block ends up dirty in

[3] While this protocol action is further optimized in DASH [28], it does not affect the reasoning in this paper.

home. During the time write-back requests and invalidations are pending, the block is in the transient state "busy" and read and write requests to the block have to be retried.

In the hardware-only directory implementation, the memory protocol engine consists of three parts implemented in hardware: a memory controller, a directory, and a state memory. The controller is responsible for processing incoming coherence requests, taking correct actions depending on the state of the memory block, and also managing the directory. By contrast, in a software-only directory protocol, the management of the directory is migrated from the complex hard-wired memory controller to software handlers executed on the compute processor. These handlers are responsible for processing coherence requests and managing the directory, which now is stored in main memory as shown in Fig. 1 and not in a special hardware directory.

Our example software-only directory protocol implementation is fairly aggressive in the sense that it uses support mechanisms to get rid of some performance limitations as proposed in [14, 15]. However, because the compute processor occupancy still shows up as the major performance obstacle in this implementation, the experimental observations we make will be further emphasized in less aggressive implementations. The first set of such support mechanisms is an interrupt and a send buffer (shaded in Fig. 1) connected to the network interface and to the second-level cache bus. They are used to trigger protocol handler invocations and post outgoing protocol requests, such as write-back requests, respectively. Another optimization is to off-load the compute processor in some common cases. By making the protocol state visible to the memory controller, which we proposed in [14, 15], all incoming requests that do not change the protocol state, such as local miss requests to clean blocks, can be directly satisfied and do not have to interrupt the compute processor.

2.2. A Framework for Reasoning about Relative Efficiency

In order to reason about the relative performance between hardware-only and software-only directory implementations, especially when latency-tolerating and -reducing techniques are used, we first present a simple model that isolates where the execution time is spent in a parallel application. In the leftmost bar of Fig. 2,

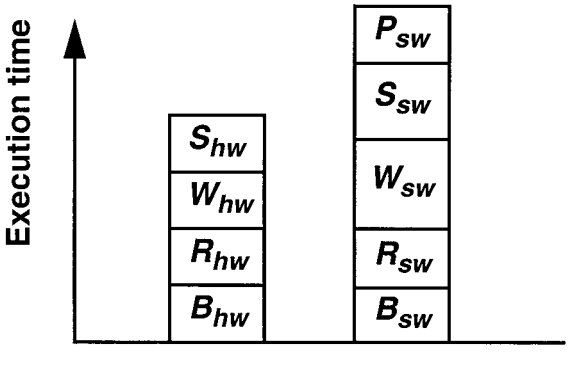

FIG. 2. Execution time breakdown for hardware-only (left) and software (right) directory protocols.

the execution time of a parallel application on a hardware-only system is broken down into the fraction of busy cycles, B_{hw}, and three stall time components: read stall (R_{hw}), write stall (W_{hw}), and synchronization stall (S_{hw}). While the read stall time is the time the processors are stalled due to read misses in the caches, the write stall time is the time the processors are stalled waiting for ownership requests to complete. Finally, the synchronization stall time is the time the processors wait for, e.g., locks and barriers. The total execution time under a hardware-only directory protocol, denoted E_{hw}, is thus $E_{hw} = B_{hw} + R_{hw} + W_{hw} + S_{hw}$.

The execution time can be split into the same components under the software-only directory protocol. In addition to the stall time components, the overhead due to protocol execution, denoted P_{sw}, exists as shown in the rightmost bar of Fig. 2. This overhead component arises from the fact that each coherence request to home interrupts the compute processor in the home node. When a software handler is invoked, the execution of it may be overlapped with other stall times and does not interrupt application execution. Otherwise, the handler execution is only partly, or in the worst case not at all, overlapped with other stall time components. In this case, the handler execution prolongs the total execution time of the application. The sum of handler execution times that are not overlapped with other stall times is referred to as *p-time* and denoted P_{sw} in Fig. 2. In [14], we found very limited opportunities to overlap protocol execution with other stall times. P_{sw} is therefore expected to directly affect the execution time. The total execution time under a software-only directory protocol, denoted E_{sw}, is thus $E_{sw} = B_{sw} + R_{sw} + W_{sw} + S_{sw} + P_{sw}$.

Throughout this paper, we use the *execution time ratio* (ETR) between hardware-only and software-only directory protocols as our primary measure of the relative performance to judge whether a certain latency-tolerating and -reducing technique will increase or decrease this ratio. Given the execution time equations, ETR is defined as

$$ETR = \frac{E_{sw}}{E_{hw}} = \frac{B_{sw} + R_{sw} + W_{sw} + S_{sw} + P_{sw}}{B_{hw} + R_{hw} + W_{hw} + S_{hw}}.$$

2.3. Discussion

As for the relative size of the various components in the two systems, we first note that $B_{hw} = B_{sw}$ for all applications because all our benchmarks use static work distribution. Second, because the software-only architecture we use in this study employs the strategies for data forwarding proposed and evaluated in [14], the read stall time components are the same; i.e., $R_{hw} = R_{sw}$. When it comes to the write stall time and the synchronization overhead, however, we found in [14] that they can be significantly higher in a software-only directory than in a hardware-only directory protocol, i.e., $W_{hw} < W_{sw}$ and $S_{hw} < S_{sw}$. Their impact together with the protocol execution overhead (P_{sw}) results in an ETR > 1 for the baseline systems. In [14] an ETR between 1.16 and 1.68 was reported.

The central question to be explored in this paper is whether the ETR can be reduced by carefully selecting among latency-tolerating and -reducing techniques proposed in the literature. By choosing techniques that can hide the stall time

components there is a hope that the hardware-only and software-only directory protocols can actually perform equally well, thus giving an advantage to the latter in terms of implementation complexity. Before we present and explore the performance consequences of the various improvement techniques, we next present the methodology that drove our experiments.

3. EXPERIMENTAL METHODOLOGY

The simulation models are built on top of the CacheMire Test Bench [2], a simulation framework and programming environment. The framework consists of multiple SPARC processors simulated at the instruction level and an architectural simulator of the multiprocessor model. The processors issue memory references to the architectural simulator, which delays the processors according to its timing model. Thus, the same interleaving as in the target system is obtained. Instruction and private data references are not fed into the architectural simulator since we assume they hit in cache and are carried out in a single cycle.

3.1. Simulation Environment and Architectural Parameters

We simulate multiprocessor architectures with 16 processor nodes according to Fig. 1, and in Table 1 we have collected the architectural parameters. Memory pages are allocated using a first-touch 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 *first-level cach*e (*FLC*); i.e., they do not incur any memory system latencies. The default memory consistency model in this study

TABLE 1

Architectural Parameters

Parameter	Value ad Comments
Block size	64 bytes
First-level (*FLC*) and second-level cache (*SLC*) sizes	8 KByte *FLC* (write-through, direct-mapped) + 256 KByte *SLC* (full inclusion, direct-mapped, lockup-free)
Write-buffer sizes	16 entries
Page size	4 KByte
Processor speed	500MHz, single-issue (1 pclock = 2ns)
FLC acces time	2 ns = 1 pclock
SLC access time (for a block)	16 ns = 8 pclocks (SRAM, interleaved)
Memory access time (for a block)	120 ns = 60 pclocks (DRAM, interleaved)
Interrupt and send buffer access times	10 ns = 5 pclocks
Bus speed (128-bits wide, split transaction)	100 MHz, 10 ns arbitration + 10 ns tranfer
Network interface speed	100 MHz, 400 Mbytes/s into and out of each node
Network latency (infinite bandwidth)	150 ns = 75 pclocks (approximately the latency in a 100-MHz mesh network with 32-bit flits)
Software handler execution time	100 pclocks + additional time as described in the text

is sequential consistency and the processor is stalled on each shared data access until it is completed. Acquires and releases are supported by a queue-based lock mechanism, similar to the one implemented in the DASH [28]. In the software-only directory protocols this mechanism is implemented by software handlers.

The network interface is responsible for routing messages between the node and the network. It also collects invalidation acknowledgments and notifies the memory-protocol engine when the last acknowledgment has arrived. The interrupt and send buffers are interfaced to the SLC bus. They have the same access time as the SLC and are accessible through memory mapped addresses. In a real system special precautions 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. The network interface speed and the network latency and bandwidth are comparable to those of the MAGIC controller and the mesh, respectively, in the Stanford FLASH [20]. The network speed (100 MHz) may seem conservative compared to the processor speed (500 MHz). However, a 500-MHz single-issue processor has the same instruction rate as a 250-MHz ideal dual-issue processor. Thus, our assumptions correspond well to the SGI Origin, which has multiple-issue processors running at 195 MHz and a network running at 100 MHz [27].

Table 2 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 as a result of contention. Contention is correctly modeled in all parts of the processor nodes. Clearly, in a software-only directory protocol, software handler invocations also prolong the request latencies.

A critical timing assumption is how many cycles we charge for the software handlers. We assume 100 pclocks as the default protocol execution time, which does not include the time to access memory, the buffers, and the directory and state information. On top of this we charge 130 ns to read or write the global state of a memory block; 2 or 160 ns for a directory access depending on whether the directory is cached or not, 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

TABLE 2

Average Read-Miss Latency Times for a Hardware-Only Implementation Assuming a Conflict-Free System

Latency numbers for read requests	1 pclock = 2ns (500 MHz)
Fill from FLC	1 pclock
Fill from SLC	8 pclocks
Fill from local memory	88 pclocks
Fill from home (clean, 2-hop)	288 pclocks
Fill from remote (dirty, 4-hop)	598 pclocks

TABLE 3

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-by-258 grid
Water–Nsquared	Molecular dynamics simulation, $O(N^2)$ algorithm	512 molecules, 3 time steps
Cholesky	Blocked sparse Cholesky factorization	tk29.0
FFT	1-D $\sqrt{n}$ six-step fast Fourier transform	64K points
Radix	Integer radix sort	256K keys, radix 1024

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.

As suggested in [15], we assume that directory entries are cached when the software handlers access them. 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 4 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 appears as follows (in C-syntax):

$$Address_{directory} = 0 \times 70000000 \mid ((Address_{data}/block\_size) * 4).$$

3.2. Benchmark Programs

In order to understand the relative performance of hardware-only and the variations of the software-only directory protocols, we use six 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 [38].

A short description of the applications together with the data set sizes we use is shown in Table 3. 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 [38].

4. EXECUTION TIME EFFECTS OF PREFETCHING

The first technique we will study is prefetching. We start with a qualitative discussion in Section 4.1 and then a quantitative evaluation of two prefetching schemes in Section 4.2.

4.1. Qualitative Evaluation

Prefetching is a latency-tolerating technique that appears in the literature as a software-controlled [3, 26, 30] as well as a hardware-based technique [6, 9, 10]. We start with the general characteristics of prefetching and then specifically discuss the schemes we use in this study. In this study we only consider nonbinding, read-shared prefetching, i.e., the block is fetched in a shared mode. We will discuss other forms of prefetching as well in Section 7.

The goal of read-shared, nonbinding prefetching is to bring data into the cache in advance so the processor encounters a cache hit instead of a miss, a so-called *useful* prefetch. The data is fetched in a shared state and is still visible to the coherence protocol, as opposed to binding prefetching where data is loaded directly into, e.g., a register. As a result, the data might be evicted from the cache, due to an invalidation or a replacement, before the processor accesses it. In this situation the processor still encounters a miss and the prefetch is *useless*. Useless prefetches also originate from the fact that the prefetch scheme might fetch blocks that the processor never accesses. All these useless prefetches both cause unnecessary network traffic and, more important to this study, *increase the occupancy* in the memory protocol engine of the home node. This occupancy is usually no problem in a hardware-only directory protocol since each prefetch occupies the controller only a short amount of time. By contrast, in a software-only directory protocol this occupancy is directly translated into protocol execution overhead in the home node, i.e., p-time increases. We will refer to the ratio between the number of useful prefetches and the total number of prefetches as the *prefetch efficiency*. Further, we will refer to the fraction of the total number of read misses removed by useful prefetches as the *coverage*.

When prefetching is applied to both hardware-only and software-only directory protocols, we are interested in whether the *ETR* will increase or not. Prefetching attacks and reduces the read stall time. In the ideal case the read stall time is equally reduced in both hardware-only and software-only directory protocols; i.e., $R'_{hw} = R'_{sw} < R_{hw} = R_{sw}$. The write and synchronization stall times may increase a little as a result of contention. However, the big difference between hardware-only and software-only protocols is p-time, which increases as a result of useless prefetches, i.e., P_{sw} increases. The important question is whether the read stall time reduction is large compared to the increase in protocol execution overhead or not and for which schemes this can be obtained.

Useless prefetches are present to various degrees in all prefetching schemes, and we consider two different prefetching schemes in this study: hardware-based adaptive sequential prefetching [10] and an ideal scheme. The first scheme we evaluate is *adaptive sequential prefetching* [10]. While the implementation details are found in [10], we here concentrate on the behavioral aspects. In sequential prefetching a fixed number of consecutive blocks, K, are prefetched for each cache miss the processor encounters. The K prefetched blocks are those directly following the missing block in the address space. However, the optimal number of blocks to prefetch on each miss varies during the execution. In adaptive sequential prefetching this shortcoming is addressed. By measuring the number of useful prefetches, i.e., the

number of prefetched blocks that the processor actually accesses before they are evicted from the cache, the scheme tunes the value of K according to the dynamic behavior of the application which maintains a high prefetch efficiency.

The second scheme we evaluate is an ideal prefetching scheme that has a fixed coverage and a fixed prefetch efficiency. At the time a processor encounters a potential cache miss, we statistically determine whether it should have been covered by a useful prefetch or actually results in a cache miss. If the miss is determined to be covered by a useful prefetch, a number of useless prefetches are generated depending on the prefetch efficiency. A potential drawback of this approach is as follows. If the prefetch efficiency is very low, e.g., below 10%, many useless prefetches are issued simultaneously, which may result in a clustering effect not present in a real scenario. However, since prefetch studies in the literature [10, 31] have reported significantly higher efficiency numbers, we believe that this effect is negligible. By varying the coverage and the prefetch efficiency in our simulations, we can cover the behavior of a large portion of prefetching schemes proposed in the literature.

4.2. Quantitative Evaluation

In this section we quantitatively evaluate how the relative performance changes between a hardware-only and a software-only directory protocol when prefetching is applied. We start in Section 4.2.1 with adaptive sequential prefetching and continue in Section 4.2.2 with the ideal scheme.

4.2.1. *Adaptive sequential perfetching.* In this section we evaluate how the relative performance between a hardware-only and a software-only directory protocol changes when adaptive sequential prefetching is applied. The execution times of the six applications are shown in Fig. 3. For each application, four bars are shown. The two left bars correspond to the execution times for a hardware-only directory protocol without (HW) and with (HW-ADAP) adaptive sequential prefetching, and the two right bars correspond to the execution times for a software-only directory protocol without (SW) and with (SW-ADAP) adaptive sequential prefetching. For each bar, the execution time is decomposed, from bottom to top and with the notations from Fig. 2, into the following components: the busy (B_x), the read stall (R_x), the write stall (W_x), and the synchronization stall time (S_x), where x is either *hw* or *sw*. Finally, for software-only directory protocols, the protocol execution overhead (P_{sw}) is shown at the top. To simplify the discussion, we will use these notations in the rest of this paper.

We first focus on the relative execution times of HW and SW in Fig. 3. By comparing the execution times, we find that the execution time ratios between SW and HW are between 1.12 (Water–Nsquared) and 1.59 (Ocean). These ETR values are in accordance with the results earlier presented in [14, 15].

Prefetching aims at reducing the read stall times, thus obtaining a shorter execution time. As we see in Fig. 3, the execution times under HW-ADAP are lower than those under HW for four of the applications. In addition, simulation results show that prefetching reduces R_{hw} with 7, 11, 1, 25, 47, and 62% for Barnes, Ocean, Water–Nsquared, Cholesky, FFT, and Radix, respectively.

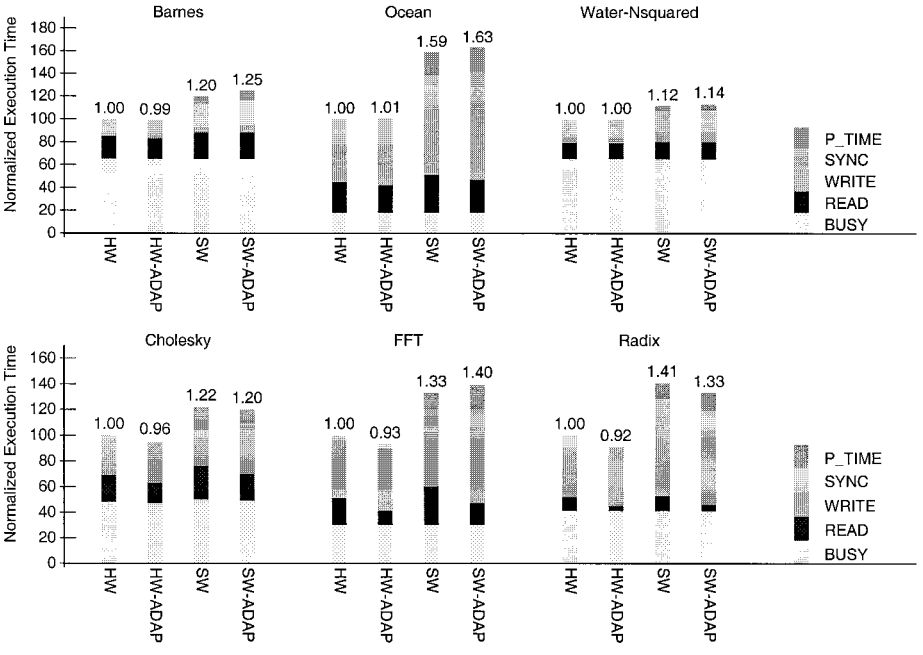

FIG. 3. Normalized execution times for hardware-only and software-only directory protocols without (HW and SW) and with adaptive sequential prefetching (HW-ADAP and SW-ADAP).

Continuing with the performance effects under the software-only directory protocol, we find by comparing the read stall times under SW and SW-ADAP that prefetching reduces R_{sw} with 11, 1, 20, 43, and 58% for Ocean, Water–Nsquared, Cholesky, FFT, and Radix, respectively. So in that respect, prefetching helps performance. For Barnes R_{sw} is the same for SW and SW-ADAP. Unfortunately, both W_{sw} and S_{sw} increase by up to 13% (Barnes) and 239% (FFT), respectively, under SW-ADAP compared to SW as a result of longer queuing delays in the home node. The average number of messages in the interrupt buffer at the time a new request is deposited in the buffer increases under prefetching. For example, for FFT the average number of messages in the buffers in the network interface upon deposit of a new message is increased from 0.09 to 3.69 when prefetching is applied.

By examining the protocol execution overhead, we observe a significant increase of P_{sw} under SW-ADAP compared to SW; P_{sw} has increased by between 7% (Radix) and 39% (FFT). This higher protocol execution overhead is expected according to the discussion in Section 4.1. The increase of P_{sw} stems from a higher number of coherence requests to home, and our simulation results show that SW-ADAP generates between 1% (Radix) and 26% (Barnes) more software handler invocations than SW.

Finally, by adding all the execution time components under SW and SW-ADAP, we conclude that the total execution time, E_{sw}, has decreased for two applications (Cholesky and Radix) and increased by up to 5% (FFT) when adaptive sequential prefetching is applied to the software-only directory protocol. As a result, the execution time ratios (ETR) between SW-ADAP and HW-ADAP are higher than those between SW and HW for all applications. We summarize the resulting ETR values in Table 4.

TABLE 4

Execution Time Ratios between Software-Only and Hardware-Only Directory Protocols without and with Adaptive Sequential Prefetching

	Barnes	Ocean	Water–Nsquared	Cholesky	FFT	Radix
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.20	1.59	1.12	1.22	1.33	1.41
$ETR = E_{sw}(\text{SW-ADAP})/E_{hw}(\text{HW-ADAP})$	1.27	1.61	1.14	1.25	1.51	1.45

One reason the adaptive sequential prefetching scheme incurs so much protocol execution overhead can be tracked to a low prefetch efficiency, which turns out to be between 33% (Barnes) and 47% (Radix). In other words, for each successful prefetch, i.e., a prefetched block that the processor accesses before the block is evicted from the cache, between one and two useless prefetches are issued. Even though useless prefetches increase contention in a hardware-only directory protocol, they can have a devastating effect on the performance of software-only directory protocols. Therefore, a high prefetch efficiency in a prefetching scheme is more important in software-only than in hardware-only directory protocols.

4.2.2. *Ideal prefetching scheme with fixed coverage and prefetch efficiency.* In order to better understand how the efficiency of prefetching schemes impacts the performance of software-only directory protocols, we have simulated an ideal prefetching scheme with a fix coverage and prefetch efficiency. We start with a default coverage and prefetch efficiency, which is based on findings in other studies, and then do a variation analysis at the end of this section. In [9], Dahlgren and Stenström evaluated hardware-based stride and sequential prefetching. They reported coverage numbers between 2 and 80% and prefetch efficiencies between 13 and 92%. Mowry evaluated software prefetching in his thesis [31] and presented coverage numbers between 75 and 98%, while the prefetch efficiency varied between 11 and 85%. As default numbers in our study, we have chosen a coverage of 50% and a prefetch efficiency of 25%, which represent a fairly conservative prefetching scheme.

The execution times of the six applications are shown in Fig. 4. For each application, four bars are shown. The two left bars correspond to the execution times for a hardware-only directory protocol without (HW) and with (HW-PRE) prefetching, and the two right bars correspond to the execution times for a software-only directory protocol without (SW) and with (SW-PRE) prefetching.

As seen in Fig. 4, the execution times under HW-PRE are lower than those under HW for all applications. In addition, simulation results show that prefetching reduces R_{hw} with between 24% (Cholesky) and 39% (Radix). Since we now have a fixed coverage, we see smaller differences in the R_{hw} reduction between different applications than for adaptive sequential prefetching.

Continuing with the performance effects under the software-only directory protocol, we find by comparing the read stall times under SW and SW-PRE that prefetching reduces R_{sw} between 23% (Cholesky) and 42% (FFT). Unfortunately,

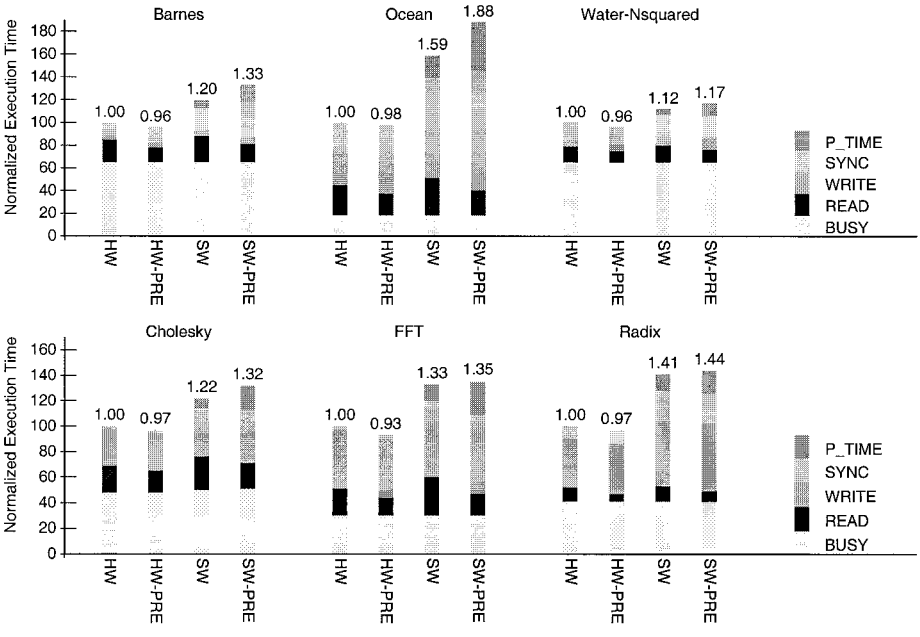

FIG. 4. Normalized execution times for harware-only and software-only directory protocols without (HW and SW) and with ideal prefetching (HW-PRE and SW-PRE).

all other stall times have increased under SW-PRE compared to SW. W_{sw} has increased by between 3% (FFT) and 22% (Ocean), S_{sw} has increased by between 2% (Radix) and 54% (Barnes), and P_{sw} has increased by between 39% (Radix) and 137% (Barnes) under SW-PRE. These stall times increase for the same reason as for adaptive sequential prefetching, i.e., a higher number of coherence requests to home, which results in higher contention in the processor nodes. By adding all the execution time components under SW and SW-PRE, we conclude that the total execution time, E_{sw}, has increased by between 2% (FFT and Radix) and 18% (Ocean) under SW-PRE. In summary, the execution time ratios (ETR) between SW-PRE and HW-PRE are higher than those between SW and HW for all applications. The resulting ETR values are shown in Table 5.

The reason ideal prefetching incurs so much protocol execution overhead is the low prefetch efficiency, just as for adaptive sequential prefetching. An interesting question is then how high coverage and prefetch efficiency are necessary for prefetching to be effective, i.e, when does the reduction of R_{sw} outweigh the increase

TABLE 5

Execution Time Ratios between Software-Only and Hardware-Only Directory Protocols without and with Ideal Prefetching

	Barnes	Ocean	Water–Nsquared	Cholesky	FFT	Radix
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.20	1.59	1.12	1.22	1.33	1.41
$ETR = E_{sw}(\text{SW-PRE})/E_{hw}(\text{HW-PRE})$	1.39	1.92	1.22	1.36	1.45	1.48

of P_{sw}? Therefore, we have also simulated coverage values between 25 and 90% and prefetch efficiencies between 25 and 90%. As a case study we present experimental results for Ocean since the effects are most pronounced for that application, but the other applications indicate the same trends.

In Fig. 5, we show the normalized execution times for the software-only directory protocol with (SW-PXX) and without (SW) prefetching, where XX is the prefetch efficiency (25, 50, 75, and 90%) for a given coverage. We start by concluding that prefetching reduces R_{sw} for all coverage and prefetch efficiency numbers, although the reduction is quite small when the coverage is only 25%. However, we observe that even though prefetching reduces R_{sw}, the performance gain is many times outweighed, or at least greatly reduced, by longer write and synchronization stall times and higher protocol execution overhead.

Looking at the results in more detail, we can see some interesting trends as the coverage and the prefetch efficiency vary. We emphasize that the exact execution times are not the most important thing to observe, but the trends are more important. First, we have observed that a prefetch efficiency below or equal to 25% results in longer or equal execution times for all applications as compared to those without prefetching. By contrast, for hardware-only protocols we have observed execution time reductions for all applications even with only 25% prefetch efficiency and a very low coverage (25%).

Second, we can see that if the coverage is very low, i.e., less than or equal to 25%, a high prefetch efficiency is not essential, as long as it is at least 50%. For a prefetch efficiency of 50% the total execution time is approximately equal (only slightly worse or slightly better) to that without prefetching. However, as the coverage increases, a high prefetch efficiency becomes more important. As shown in Fig. 5, the largest contention problem arises when the coverage is high and the

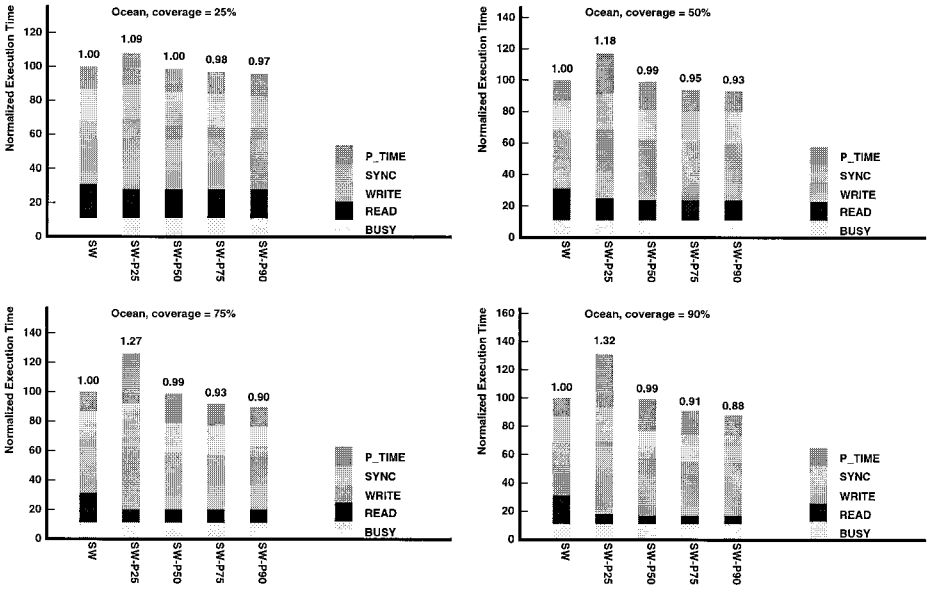

FIG. 5. Normalized execution times of the software-only directory protocol with (SW-PXX) and without (SW) prefetching for various coverage values and prefetch efficiencies for Ocean.

prefetching efficiency is low. For example, with 25% coverage, the execution time for SW-P25 is only 9% worse than for SW, but with 90% coverage SW-P25 has 32% longer execution time than SW. As a result, with a high coverage, there is a large performance gain with a high prefetch efficiency, but also a high risk for low performance if the prefetch efficiency turns out to be low. In other words, if we use a prefetching scheme with high coverage, we had better make sure it also has a high prefetch efficiency.

In summary, even though a prefetching scheme is efficient when applied under a hardware-only directory protocol, it might increase the execution time of a software-only directory protocol if not carefully applied. Since software-only directory protocols are more sensitive to useless prefetches than hardware-only directory protocols, a low prefetch efficiency can have a devastating effect on the performance of software-only directory protocols, especially if the coverage is high. For example, adaptive sequential prefetching does not provide any consistent performance improvement for software-only directory protocols and the overall usefulness of it is questionable in software-only directory protocols. Therefore, other prefetching schemes with higher prefetch efficiencies seem to be more promising alternatives to consider.

5. EXECUTION TIME EFFECTS OF MIGRATORY OPTIMIZATION

In the previous section we studied prefetching and found that the main obstacle of prefetching in the context of software-only directory protocols was the increased p-time. By contrast, we now discuss a technique that *decreases* p-time when it is applied. We start by describing the technique and the expected impact on the *ETR*, and then we present our experimental evaluation.

5.1. Qualitative Evaluation

Migratory sharing [18] is a program behavior not uncommon in parallel applications, e.g., data accessed in critical sections and by short read–modify–write sequences such as "`i=i+1`" exhibit migratory sharing. For example, consider the following scenario: One processor first reads a data block and then it modifies the block; i.e., it obtains exclusive ownership of the block. Another processor then reads and modifies the block in the same way, and thus, the block *migrates* around among the processors. Note that migratory data blocks are referenced by only one processor at the same time but by many processors in the long run.

In a system with a write-invalidate protocol each migration of the block between two processors incurs two global actions; first a read-miss request and then an ownership request. In a sequential consistent system, both these actions stall the processor, resulting in both read and write stall times. However, two studies [8, 36] independently came up with the same solution to detect migratory blocks and optimize the coherence protocol for them. This *migratory optimization technique* dynamically detects migratory blocks at the home node, which sees all read-miss and ownership requests, by recognizing two subsequent read–write sequences by two different processors. The coherence protocol then handles migratory blocks

with a single read-exclusive request instead of one read-miss and one ownership request, thus avoiding the ownership request and the associated protocol actions and their latencies.

The performance gain achieved by the migratory optimization technique is the removal of global ownership requests for migratory blocks. As a result, the write stall time is reduced; i.e., W_{hw} and W_{sw} in Fig. 2 are reduced. More importantly, since the number of ownership requests is reduced one should expect the additional gain for software-only protocols of reducing protocol execution overhead (P_{sw}). The question is whether this reduction can actually make software-only directory protocols perform as well as hardware-only directory protocols, i.e., making *ETR* close to one. We study this performance tradeoff next.

5.2. *Quantitative Evaluation*

We will now present how the migratory optimization technique impacts the execution time components. In Fig. 6, we show the resulting execution times when migratory optimization is applied to hardware-only and software-only directory protocols, referred to as HW-M and SW-M, respectively. We first conclude that the migratory optimization reduces execution times for all applications under both hardware-only and software-only directory protocols. In order to understand the effects the migratory optimization has on the different execution time components, we go through each of them starting with the write stall time component.

By looking at the write stall times in Fig. 6, we see that both W_{hw} and W_{sw} are slightly reduced for all applications. The only two applications where the write stall time is significantly reduced when the migratory optimization technique is applied

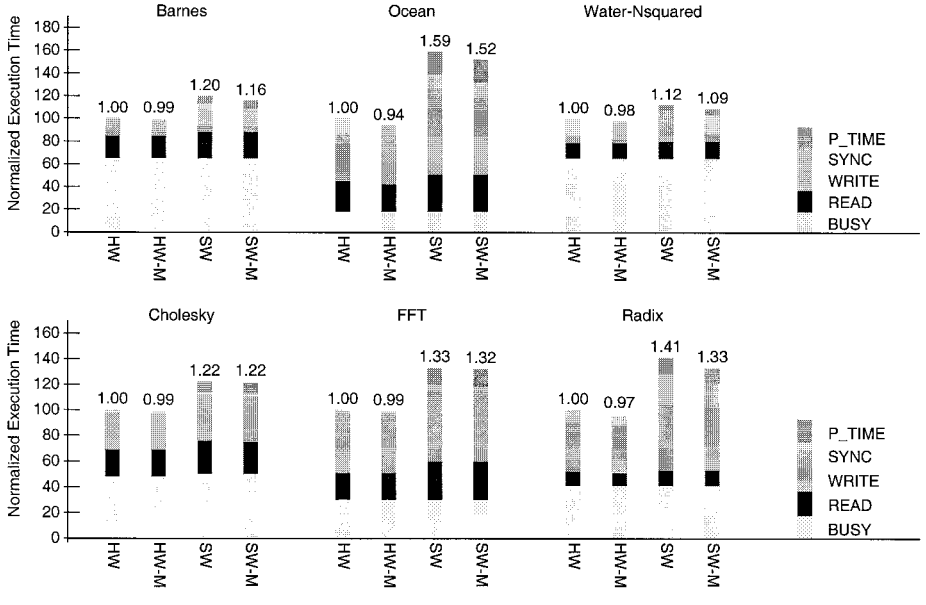

FIG. 6. Normalized execution times for hardware-only and software-only directory protocols without (HW and SW) and with (HW-M and SW-M) migratory optimization.

are Barnes and Water–Nsquared. This result is expected from the discussion in Section 5.1 and in accordance with the results presented in [36]. By looking at the write stall time reduction in more detail, we find that W_{hw} is reduced by 11 and 31% for Barnes and Water–Nsquared, respectively, while the corresponding numbers for W_{sw} are 12 and 25%, respectively. W_{sw} and W_{hw} are reduced as a result of shorter queuing delays in home; for SW-M the lower number of coherence requests reduces the average size of the interrupt buffer. For the other applications, only small decreases in the write stall times are observed. In total, this corresponds well to the application behavior; Water–Nsquared is the only application that has a significant amount of migratory sharing. In a previous study [17] where we used the older SPLASH suite, we observed significantly larger gains from the migratory optimization technique for two applications (Water and MP3D), which have much more migratory sharing.

Continuing with the synchronization stall times, we observe that they are also reduced when the migratory optimization is applied. This effect, which is most pronounced for Barnes (only for SW-M), Ocean, FFT, and Radix, arises mainly because of more efficient barrier synchronizations. In our barrier implementation, a counter variable is used to keep track of the current number of processors waiting at the barrier. This counter variable exhibits migratory sharing and, of course, benefits from the migratory optimization.

The next execution time component to examine is the protocol execution overhead, P_{sw}, which is reduced by 11% for Water–Nsquared. This reduction stems from a significantly lower number of coherence interrupts in the home nodes as we predicted in Section 5.1. For Water–Nsquared, the number of coherence interrupts has decreased by 9%. For the other applications P_{sw} is virtually unaffected as expected. Finally, both R_{hw} and R_{sw} are virtually unaffected ($\leqslant 2\%$) for all applications.

In Table 6, we show the resulting execution time ratios of SW to HW and of SW-M to HW-M, respectively. Using the migratory optimization results in lower *ETR* for applications with migratory sharing (Barnes and Water–Nsquared). The lower *ETR* results from a relatively larger reduction of W_{sw} and S_{sw} than of W_{hw} and S_{hw} and from a reduction of P_{sw}. For Radix, the *ETR* is lower as a result of a relatively larger reduction of S_{sw} than of S_{hw}. For Cholesky and FFT, we find that the *ETR* is virtually the same with and without the migratory optimization. Finally, for Ocean the *ETR* increases, mainly as a result of a relatively smaller reduction of S_{sw} than S_{hw} when migratory optimization is applied.

TABLE 6

Execution Time Ratios between Software-Only and Hardware-Only Directory Protocols without and with Migratory Optimization

	Barnes	Ocean	Water–Nsquared	Cholesky	FFT	Radix
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.20	1.59	1.12	1.22	1.33	1.41
$ETR = E_{sw}(\text{SW-M})/E_{hw}(\text{HW-M})$	1.17	1.62	1.11	1.23	1.33	1.37

In summary, we have found the migratory optimization effective in reducing both the write stall time and the protocol execution overhead for applications with migratory sharing. Since the write and synchronization stall times are relatively more reduced under SW-M than under HW-M, the execution time ratio decreases for three of the applications. Since our software-only directory implementation is fairly aggressive, the effects of migratory optimization are rather small. We expect the relative benefits of the migratory optimization to increase in less aggressive implementations. Further, the migratory optimization has a potential to reduce the synchronization stall times also for other applications as a result of more efficient barrier synchronizations.

6. EXECUTION TIME EFFECTS OF RELEASE CONSISTENCY

The third technique we will study in this paper is Release Consistency, which is a technique that does not affect the protocol execution time. We have seen that it is possible to achieve virtually the same read stall times for SW and HW, but the software handler execution time will appear on the critical memory access path for ownership acquisitions. The processor in the home node must examine the directory before sending invalidations, and as a result, the software handler execution time cannot be removed from the write stall time seen by local. By contrast, under relaxed memory consistency models, e.g., release consistency, the write latency can be overlapped by application execution as long as synchronizations are not encountered, without changing the number of global write actions. In this section, we will show how this impacts the execution time ratio, starting with a qualitative discussion of the expected effects.

6.1. Qualitative Evaluation

Under sequential consistency, which is our default memory consistency model, the processor stalls on each access to shared data in order to enforce a global order of accesses. This is a severe restriction and can be relaxed. Under relaxed memory consistency models, ordering is only enforced on special, hardware-recognizable synchronization primitives. One of the most relaxed consistency models is *Release Consistency* (*RC*) [13]. In RC, one distinguishes between *acquires* (acquiring a lock or a flag) and *releases* (releasing a lock or a flag). RC specifies that the processor is not allowed to proceed after an acquire before the acquire has completed and that a release is not allowed to be issued until all preceding requests have completed. The most important implication of RC in our framework is that the processor does not stall on write and release requests; i.e., all write stall time can be hidden and the synchronization stall time might decrease.

Relating the expected performance effects of RC to the bars in Fig. 2, we note that $W_{hw} < W_{sw}$ under sequential consistency. Since $W'_{hw} = W'_{sw} = 0$ under RC, we expect the software-only directory protocol to gain relatively more from RC than the hardware-only directory protocol does. However, since relaxed memory consistency models make the processors execute faster, we would expect that the rate of interrupts increases. This can potentially make the read and synchronization stall

time components longer, which can offset the gain of the write stall time reduction. Therefore, we expect that a slight increase in S_{hw} and S_{sw} as well as R_{hw} and R_{sw} can occur as a result of higher contention within the processor nodes. Finally, since RC is not expected to change the number of coherence requests in the system, we expect the protocol execution overhead, i.e., P_{sw}, to be unaffected. The question to be addressed next is how these factors affect the *ETR*.

6.2. *Quantitative Evaluation*

In this section we experimentally evaluate the relative performance of hardware-only and software-only directory protocols when release consistency is applied. In Fig. 7, we show the relative execution times of hardware-only and software-only directory protocols under sequential consistency (HW and SW, respectively) and release consistency (HW-RC and SW-RC, respectively). The execution times are normalized to the execution time of a hardware-only directory protocol under sequential consistency.

We start with a comparison between the execution times of HW and HW-RC and between the execution times of SW and SW-RC. The results presented in Fig. 7 show that release consistency gives a large and consistent performance improvement for both hardware-only and software-only directory protocols, which is consistent with results presented in an earlier study [14].

By comparing the protocol execution overhead for SW and SW-RC, we see that the intuition from Section 6.1 seems to be correct; since the numbers of coherence requests are the same under sequential and release consistency, P_{sw} should not be affected. As the results in Fig. 7 show, P_{sw} is virtually unaffected for most applications,

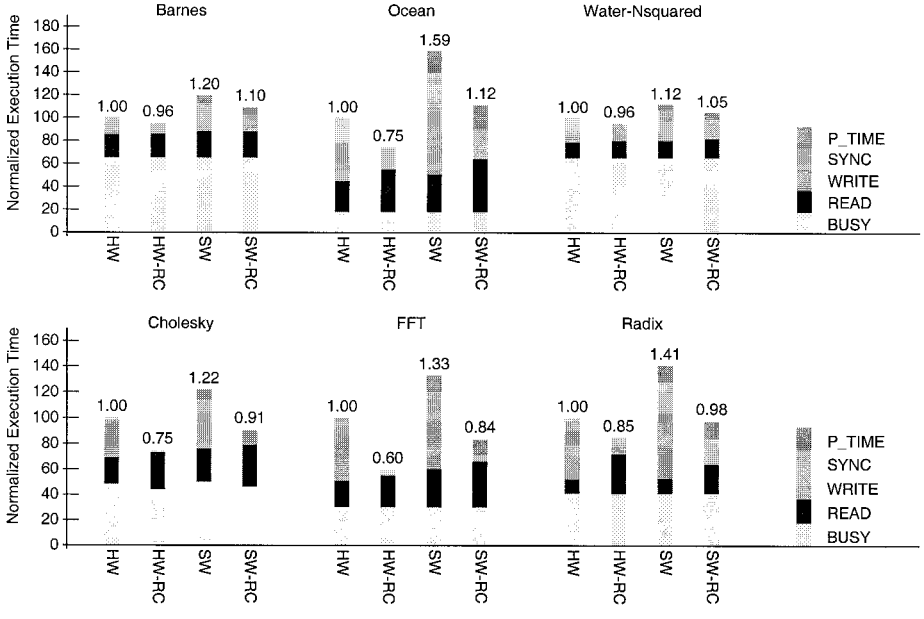

FIG. 7. Normalized execution times for hardware-only and software-only directory protocols under sequential consistency (HW and SW) and under release consistency (HW-RC and SW-RC).

TABLE 7

Execution Time Ratios between Software-Only and Hardware-Only Directory Protocols under Sequential Consistency and Release Consistency

	Barnes	Ocean	Water–Nsquared	Cholesky	FFT	Radix
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.20	1.59	1.12	1.22	1.33	1.41
$ETR = E_{sw}(\text{SW-RC})/E_{hw}(\text{HW-RC})$	1.15	1.49	1.09	1.21	1.40	1.15

which confirms our intuition. For Water–Nsquared and Cholesky, P_{sw} has increased by 18 and 24%, respectively. The reason is that a smaller portion of the total protocol execution time is covered by other stall times under SW-RC than under SW.

As seen in Fig. 7, release consistency removes the write stall time for all applications. Under the hardware-only directory protocol, this is achieved with virtually no increase in read and synchronization stall times for two of the applications. For Ocean, Cholesky, FFT, and Radix R_{hw} increases by 33, 28, 17, and 179%, respectively. Under the software-only directory protocol the read stall time is increased for the same four applications with 39, 25, 15, and 85%, respectively. This stems mainly from the following reasons: higher contention in the nodes and longer delays in the write buffers. As an example we consider Ocean. Under sequential consistency, an *FLC* read miss gets served by the *SLC* almost at once, while under release consistency it has to wait in average 20 pclocks in HW-RC and 30 pclocks in SW-RC. The longer queuing delays under RC occur since the second-level write buffer with its 16 entries is filled up, and thus blocks the *SLC*. Further, in a software-only directory protocol, each global write request takes a longer time than in a hardware-only directory protocol. As a result, the second-level write buffer is filled up more often in a software-only directory protocol.

The resulting execution time ratios between SW and HW, and between SW-RC and HW-RC are presented in Table 7. For all applications except FFT, the *ETR* decreases when release consistency is applied. However, since P_{sw} is either unaffected or higher, P_{sw} constitutes a larger portion of the total execution time under release consistency than under sequential consistency.

The results presented in this section show that release consistency hides all write penalty for both hardware-only and software-only directory protocols. The execution time ratio between software-only and hardware-only directory protocols was found to decrease for five of the applications when release consistency is applied.

7. DISCUSSION AND GENERALIZATIONS

In this study, we have simulated two models of a shared-memory multiprocessor, one with a hardware-only directory protocol and one with a software-only directory protocol. Even though our quantitative results only apply to these models, it is possible to get an intuition about the relative performance between other classes of systems as we show in the following.

We have only considered read-shared prefetching in this study; i.e., the block is prefetched in a shared mode. An alternative is to allow read-exclusive prefetching [30]; i.e., when a block is prefetched, it is obtained in an exclusive mode. A successful read-exclusive prefetch not only reduces the read stall time, possibly to zero, but can also remove coherence actions associated with a separate ownership acquisition. Thus, the total number of coherence requests is expected to decrease when read-exclusive prefetching is used, and as a result, the protocol execution overhead is also expected to decrease. Our observations from Sections 4 to 6 suggest that techniques that reduce the number of protocol invocations are expected to make software-only directory protocols relatively more competitive with hardware-only directory protocols.

In Section 4.2, we found that a high prefetch efficiency is important under software-only directory protocols, but how do proposed schemes behave? Dahlgren *et al.* have evaluated hardware-based sequential prefetching [10] and stride prefetching [9]. For sequential prefetching, the coverage is between 4 and 88%, and the prefetch efficiency ranges from about 30 up to 92% in the best case. Continuing with stride prefetching, coverage values range from 2 up to 74% and prefetch efficiencies from 13 to 90%. In software-controlled prefetching schemes [3, 26, 30, 31], the compiler inserts special prefetch instructions into the code based on static program information. For example, the software prefetching schemes presented in [31] have high coverage numbers (between 75 and 98%). Unfortunately, the prefetch efficiency varies from as low as 11 up to 85%. In software-only directory protocols, all these schemes may result in low performance gains, if any at all. The high number of useless prefetches as a result of low prefetch efficiency for some applications may result in such serious performance drawbacks that these schemes are questionable in the general case. However, we speculate that software-controlled prefetching schemes can have a larger potential to increase the performance of software-only directory protocols since they can exploit knowledge about the application behavior, and thus they can more selectively insert prefetches in the code.

An alternative to adaptive sequential prefetching is stride prefetching [6, 9]. Since stride prefetching is more selective than adaptive sequential prefetching when issuing prefetches, it can potentially be better than adaptive sequential prefetching for a software-only directory protocol. Unfortunately, the results in [9] show that the most common stride is one, and then stride and adaptive sequential prefetching behave similarly in terms of read stall time reduction.

In [22], Holt *et al.* evaluate how the occupancy of the directory controller, the network latency, and the bandwidth impact the performance of distributed shared memory systems. They evaluate a range of controller implementations, from a general purpose coprocessor on the I/O-bus to hardwired controllers, and show that the controller occupancy can be a major source of contention, especially when network latencies are small. They do not include the case where the controller functionality is implemented as software handlers executed on the compute processor as we do in this study. Instead, they evaluate how prefetching impacts on the controller occupancy and performance. They found, as we do, that the occupancy in the home node can be a performance bottleneck when prefetching is used.

Migratory sharing is quite common in many applications beyond numerically intensive ones such as in SPLASH. Recently, Ranganathan *et al.* [33] showed that migratory sharing is a dominant performance bottleneck in OLTP workloads on multiprocessors. For such workloads, we would expect that software-only protocol implementations augmented with migratory optimization techniques perform quite well in comparison with hardware-centric implementations.

Multithreading is a latency-tolerating technique used in, e.g., the MIT Alewife [1]. It has been shown to be effective in hiding processor stall times for accesses that require global actions by switching to another thread of computation [19]. However, since multiple threads run on the same processor, the number of global actions originating from each processor is likely to increase. As a result, the protocol execution overhead in a software-only directory protocol is expected to increase. Therefore, building on our derived framework from Section 4, we expect the execution time ratio between software-only and hardware-only directory protocol to increase when multithreading is used.

A limitation of our study is that we consider systems built from single-processor nodes. A more cost-effective approach would be to use symmetric multiprocessors (SMPs) as building blocks to design large-scale systems. The trade-off between a P-processor system of single-processor nodes and K SMPs with P/K processors each is essentially that the number of remote misses is reduced in the clustered system. Assume that each processor generates M misses evenly distributed over the clusters. Thus the total number of misses in the system is $M*P$. For each cluster $M*P/K$ misses can be served locally, thus totally $M*P*(K-1)/K$ misses require service in another cluster. Assuming that each processor services equally many requests, it turns out that each processor responds to $(M*P*(K-1)/K)/P = M*(K\text{-}1)/K$ misses. Consider, for example, a system with 16 processors. If each cluster only has one processor, each processor has to respond to $M*15/16$ misses. On the other hand, if the processors are organized into four clusters each processor only has to respond to $M*3/4$ misses, i.e., about 19% fewer misses for each processor. Thus, the compute processor occupancy problem is smaller in a clustered system but may still pose a problem. Indeed, Karlsson and Stenström showed in [24] that the probability of finding an idle processor that can handle the coherence interrupt increases in a cluster-based multiprocessor. Another way to reduce the compute processor occupancy problem is to use simultaneous multithreading [37]. Then the software handler execution could be integrated in the processor's instruction stream and would be expected to result in a smaller protocol handler overhead.

Two recent approaches using a software protocol to maintain coherence between multiprocessor nodes are SoftFLASH [12] and MGS [39]. In both these systems they assume powerful multiprocessor clusters connected by a network. Coherence among the clusters is maintained by a software-based protocol forming a distributed virtual shared-memory system. While they use a relaxed memory consistency model in both SoftFLASH and MGS, they do not evaluate how much performance gain is achieved by using a relaxed model compared to a more restrictive model. In addition, as shown in this study, the controller occupancy can have a devastating impact on performance in such systems and reduce the positive effects of relaxed models. Further, none of the studies evaluate the effects of prefetching or

migratory optimization. Karlsson and Stenström proposed to include prefetching techniques in an SVM system on top of a network of SMP clusters [25]. The prefetching schemes managed to cut the overall execution times to some extent but it is not clear how much the protocol overhead limited further gains. More recently, Mowry *et al.* [32] studied compiler-controlled prefetching in SVM systems. While prefetching helped cut the memory stall times, the protocol overhead was quite substantial. Our study clearly demonstrates that SVM systems could be substantially improved by avoiding expensive protocol operations in latency-tolerating techniques such as prefetching.

8. CONCLUSIONS

In both software-only directory protocols and hardware-only directory protocols performance is often limited by processor stall times due to memory accesses. In addition to these stall times, the total execution time for a software-only directory protocol might be prolonged because of protocol execution overhead resulting from the handler invocations. To cope with these latencies, many latency-tolerating and -reducing techniques have been proposed and evaluated in the context of hardware-only directory protocols. However, the effectiveness of such techniques in the context of software-only directory protocols has been unexplored.

In this study we have evaluated how the relative performance between software-only and hardware-only directory protocols is affected when different latency-tolerating and -reducing techniques are applied. We developed a framework for reasoning about their expected relative efficiencies. Such techniques can be divided into three classes depending on how they impact the protocol execution overhead in software-only directory protocols; a technique increases, decreases, or does not affect the protocol execution overhead. As representative examples of the three classes we have chosen prefetching, a migratory optimization technique, and finally, release consistency as techniques that increase, decrease, and do not affect the protocol execution overhead, respectively.

Based on architectural simulations we have found that software-only directory protocols are more sensitive to useless prefetches than hardware-only directory protocols are. Each useless prefetch, i.e., a prefetch that fetches a block that is evicted from cache before the processor accesses it, incurs protocol execution overhead in the node where the memory block is allocated which potentially prolongs the total execution time. Our results show that even though prefetching reduces the read stall time in a software-only directory protocol, the execution time may be prolonged due to useless prefetches. The results for adaptive sequential prefetching show that even though the read stall time is reduced by between 1 and 62% in a software-only directory protocol, the execution time can be prolonged by up to 5% mainly due to too many useless prefetches. Therefore, greater attention must be turned to the prefetch efficiency when choosing a prefetch scheme for a software-only than when choosing one for a hardware-only directory protocol. Using an ideal prefetching scheme with fixed coverage and prefetch efficiency, we found that a prefetch efficiency of at least 50% is essential for a prefetching scheme to be effective together with software-only directory protocols.

In contrast, software-only directory protocols generally gain relatively more than hardware-only directory protocols when techniques that reduce the number of coherence actions are applied. This fact was confirmed with the migratory optimization technique. For three of six applications, the execution time ratio between software-only and hardware-only directory protocols decreases when migratory optimization is used. The effects of migratory optimization are rather small since our software-only directory implementation is fairly aggressive and the SPLASH-2 applications have very little migratory sharing. We expect to see larger benefits of the migratory optimization in less aggressive software-only directory implementations and for applications with more migratory sharing, e.g., OLTP applications [33].

Release consistency, a technique which does not affect the protocol execution overhead, manages to hide all write stall time for both software-only and hardware-only directory protocols. Since the write stall time is longer for software-only than for hardware-only directory protocols, the performance difference between them usually decreases when release consistency is applied. In total, for five of six applications this fact results in a lower execution time ratio between software-only and hardware-only directory protocols when release consistency is applied. However, since release consistency does not reduce the protocol execution overhead, it becomes a relatively larger portion of the total execution time.

Overall, this study shows that the efficiency of a latency-tolerating technique depends not only on how well it tolerates the latency as seen by the local node, but also on how it interacts with the node where a memory block is allocated. Therefore, more care has to be taken when choosing an appropriate latency-tolerating and -reducing technique for software-only directory protocols than when choosing for hardware-only directory protocols. Finally, the presented framework is expected to be useful to reason about the expected efficiencies of other latency-tolerating and -reducing techniques than the ones we have studied in detail in this paper.

ACKNOWLEDGMENTS

This research was supported in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under Contract P855. A part of this work was carried out while the authors were at the Department of Computer Engineering at Lund University.

REFERENCES

1. A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, The MIT Alewife machine: Architecture and performance, *in* "Proc. 22nd Int'l Symp. Computer Architecture," pp. 2–13, June 1995.
2. M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, The CacheMire test bench—A flexible and effective approach for simulation of multiprocessors, *in* "Proc. 26th Ann. Simulation Symp.," pp. 41–49, March 1993.
3. D. Callahan, K. Kennedy, and A. Porterfield, Software prefetching, *in* "Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)," pp. 40–52, April 1991.

4. L. M. Censier and P. Feautrier, A new solution to coherence problems in multicache systems, *IEEE Trans. Comput.* **C-27**, 12 (December 1978), 1112–1118.
5. D. Chaiken and A. Agarwal, Software-extended coherent shared memory: Performance and Cost, *in* "Proc. 21st Int'l Symp. Computer Architecture," pp. 314–324, April 1994.
6. T-F. Chen and J.-L. Baer, Effective hardware-based data prefetching for high-performance processors, *IEEE Trans. Comput.* **C-44**, 5 (May 1995), 609–623.
7. D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton, StarT-ng: Delivering seamless parallel computing, *in* "Proc. EURO-PAR'95," Lecture Notes in Computer Science, No. 966, pp. 101–116, Springer-Verlag, Berlin, August 1995.
8. A. L. Cox and R. J. Fowler, Adaptive cache coherency for detecting migratory shared data, *in* "Proc. 20th Int'l Symp. Computer Architecture," pp. 98–108, May 1993.
9. F. Dahlgren and P. Stenström, Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors, *in* "Proc. First IEEE Symp. High Performance Computer Architecture," pp. 68–77, January 1994.
10. F. Dahlgren, M. Dubois, and P. Stenström, Sequential hardware prefetching in shared-memory multiprocessors, *IEEE Trans. Parallel Distrib. Systems* **4**, 7 (July 1995), 733–746.
11. M. Dubois, J. Skeppstedt, and P. Stenström, Essential misses and memory traffic in coherence protocols, *J. Parallel Distrib. Computing* **29**, 2 (September 1995), 108–125.
12. A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory, *in* "Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)," pp. 210–220, October 1996.
13. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, *in* "Proc. 17th Int'l Symp. Computer Architecture," pp. 15–26, May 1990.
14. H. Grahn and P. Stenström, Efficient strategies for software-only directory protocols in shared-memory multiprocessors, *in* "Proc. 22nd Int'l Symp. Computer Architecture," pp. 38–47, June 1995.
15. H. Grahn and P. Stenström, "Architectural Support for an Efficient Implementation of a Software-Only Directory Cache Coherence Protocol," Technical Report 213, Department of Computer Engineering, Lund University, June 1995.
16. H. Grahn and P. Stenström, An adaptive update-based cache coherence protocol for reduction of miss rate and traffic, *J. Parallel Distrib. Comput.* **39**, 2 (December 1996), 168–180.
17. H. Grahn and P. Stenström, Relative performance of hardware- and software-only directory protocols under latency-tolerating and -reducing techniques, *in* "Proc. 11th Int'l Parallel Processing Symp.," pp. 500–506, April 1997.
18. A. Gupta and W.-D. Weber, Cache invalidation patterns in shared-memory multiprocessors, *IEEE Trans. Comput.* **41**, 7 (July 1992), 794–810.
19. A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, Comparative evaluation of latency reducing and tolerating techniques, *in* "Proc. 18th Int'l Symp. Computer Architecture," pp. 254–263, May 1991.
20. M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, The performance impact of flexibility in the Stanford FLASH multiprocessor, *in* "Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)," pp. 274–285, October, 1994.
21. M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, Cooperative shared memory: Software and hardware for scalable multiprocessors, *ACM Trans. Comput. Systems* **11**, 4 (November 1993), 300–318.
22. C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy, "The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors," Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

23. L. Iftode and J. P. Singh, Shared virtual memory: Progress and challenges, *Proc. of the IEEE* **87**, 3 (March 1999), 498–507.
24. M. Karlsson and 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)," pp. 4–13, February 1996.
25. M. Karlsson and P. Stenström, Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared memory systems, *J. Parallel Distrib. Comput.* **43**, 2 (June 1997), 79–93.
26. A. C. Klaiber and H. M. Levy, An architecture for software-controlled data prefetching, *in* "Proc. 18th Int'l Symp. Computer Architecture," pp. 43–53, May 1991.
27. J. Laudon and D. Lenoski, The SGI origin: A CC-NUMA highly scalable server, *in* "Proc. 24th Int'l Symp. on Computer Architecture," pp. 241–251, June 1997.
28. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, The DASH prototype: Logic overhead and performance, *IEEE Trans. Parallel Distrib. Systems* **4**, 1 (January 1993), 41–61.
29. T. Lovett and R. Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, *in* "Proc. 23rd Int'l Symp. on Computer Architecture," pp. 308–317, June 1996.
30. T. Mowry and A. Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors, *J. Parallel Distrib. Comput.* **12**, 2 (June 1991), 87–106.
31. T. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," Ph.D. thesis, Stanford University, March 1994.
32. T. Mowry, C. Chan, and A. Lo, Comparative evaluation of latency tolerance techniques for software distributed shared-memory, *in* "Proc. Fourth Int. Conf. on High-Performance Computer Architecture," pp. 300–311, 1998.
33. P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, Performance of database workloads on shared-memory systems with out-of-order processors, *in* "Proc. 8th Int.Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)," pp. 307–318, October 1998.
34. S. K. Reinhardt, J. R. Larus, and D. A. Wood, Tempest and typhoon: User-level shared-memory, *in* "Proc. 21st Int'l Symp. Computer Architecture," pp. 325–336, April 1994.
35. D. Scales, K. Gharachorloo, and 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)," pp. 174–185, October 1996.
36. P. Stenström, M. Brorsson, and L. Sandberg, An adaptive cache coherence protocol optimized for migratory sharing, *in* "Proc. 20th Int'l Symp. Computer Architecture," pp. 109–118, May 1993.
37. D. M. Tullsen, S. J. Eggers, and H. M. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, *in* "Proc. 22nd Int'l Symp. Computer Architecture," pp. 392–403, June 1995.
38. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, *in* "Proc. 22nd Int'l Symp. Computer Architecture," pp. 24–36, June 1995.
39. D. Yeung, J. Kubiatowicz, and A. Agarwal, MGS: A multigrain shared memory system, *in* "Proc. 23rd Int'l Symp. Computer Architecture," May 1996.

HÅKAN GRAHN is an assistant professor of computer engineering at the University of Karlskrona/Ronneby. He received a M.Sc. in computer science and engineering in 1990 and a Ph.D. in computer engineering in 1995, both from Lund University. His main interests are computer architecture, shared-memory multiprocessors, cache coherence, and performance evaluation. He has authored and co-authored more than 20 papers on these subjects. He received the Best Paper Award at the PARLE'94 (Parallel Architectures and Languages, Europe) conference. Currently he is head of department for the Department of Software Engineering and Computer Science. Dr. Grahn is a member of the ACM and the IEEE Computer Society. For more information, please refer to http://www.ipd.hk-r.se/ ~ nesse/.

PER STENSTRÖM has been a professor of computer engineering with a chair in computer architecture at Chalmers University of Technology since 1995. He was previously on the faculty of Lund University where he also received his M.S. in electrical engineering and a Ph.D. in computer engineering in 1981 and 1990, respectively. Dr. Stenström's research interests are in computer architecture and real-time systems in general with an emphasis on design principles and design methods for multiprocessor systems including software as well as hardware design issues. He has contributed to more than 50 scientific papers on multiprocessor design principles and authored two textbooks on computer architecture. As a visiting scientist, he has participated in major multiprocessor architecture research projects at Carnegie–Mellon, Stanford University, and University of Southern California. He is on the editorial board of the *Journal of Parallel and Distributed Computing* (JPDC) and guest edited a special issue of *IEEE Computer* in 1996 on shared-memory multiprocessing and a special issue of *Proceedings of the IEEE* on distributed shared memory systems in 1999. He has also been a member of numerous program committees for computer architecture and parallel processing conferences. Dr. Stenström is a senior member of the IEEE and a member of the IEEE Computer Society. For more information please refer to http://www.ce.chalmers.se/ ~ pers.