Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors

Fredrik Dahlgren, *Member*, *IEEE Computer Society*,
Michel Dubois, *Senior Member*, *IEEE*, and Per Stenström, *Senior Member*, *IEEE*

Abstract—We evaluate three extensions to directory-based cache coherence protocols in shared-memory multiprocessors. These extensions are aimed at reducing the penalties associated with memory accesses and include a hardware prefetching scheme, a migratory sharing optimization, and a competitive-update mechanism. Since each extension targets distinct components of the read and write penalties, they can be combined effectively. This paper identifies the combinations yielding the best performance gains and cost trade-offs in the context of a class of cache-coherent NUMA (Non-Uniform Memory Access) architectures. Detailed architectural simulations of a multiprocessor with single-issue, statically scheduled CPUs, using five benchmarks, show that the protocol extensions often provide additive gains when they are properly combined. For example, the combination of prefetching with the competitive-update mechanism speeds up the execution by nearly a factor of two under release consistency. The same speedup is obtained under sequential consistency by combining prefetching with the migratory sharing optimization. This paper shows that a basic write-invalidate protocol augmented by appropriate extensions can eliminate most memory access penalties without any support from the programmer or the compiler.

Index Terms—Shared-memory multiprocessors, cache-coherence protocols, prefetching, competitive-update protocols, write caches, performance evaluation.

1 Introduction

Private caches in conjunction with directory-based, write-invalidate protocols are essential, but not sufficient, to cope with the high memory latencies of large-scale shared-memory multiprocessors. Therefore, many studies have focused on techniques to tolerate or reduce the latency. In [13], Gupta et al. compared three approaches to tolerate latency: relaxed memory consistency models, software-based prefetching, and multithreading. Whereas the relaxation of the memory consistency model improves the performance of all applications uniformly, the effects of prefetching and multithreading vary widely across the application suite. Overall, these techniques are effective, but they also have negative implications for software developments because the programmer and the compiler have to worry about complex issues such as data-race detection, cache miss identification, and increased concurrency.

In order to hide the latency between the processors and the rest of the memory system or to reduce its impact on performance, we propose a different approach, namely tuning the cache protocol by adding a few simple extensions. Since these extensions are completely hardware-based, they are transparent to the software. Moreover, they add only marginally to overall system complexity while still providing a significant performance advantage when applied separately. We focus on three extensions: *adaptive sequential prefetching* [4], [5], *migratory sharing optimization* [3], [20], and *competitive-update mechanisms* [7], [12].

Adaptive sequential prefetching cuts the number of read misses by fetching a number of consecutive blocks into the cache in anticipation of future misses. The number of prefetched blocks is adapted according to a dynamic measure of prefetching effectiveness which reflects the amount of spatial locality at different times. This simple scheme relies on three modulo-16 counters per cache and two extra bits per cache line, and it removes a large number of cold, replacement, and (sometimes) true sharing misses. Moreover, as opposed to simply adopting a larger block size, it does not affect the false sharing miss rate [4]. The migratory sharing optimization targets a common sharing pattern in which a block is read and modified by different processors in turns. This optimization is aimed at the write penalty[1] for migratory blocks, a major component of the overall penalty under sequential consistency. Two independent studies [3], [20] have shown that this optimization is very successful in many cases and requires only minor modifications to the cache coherence protocol.

The above two protocol extensions are fairly ineffective when it comes to coherence misses. It is also well-known [9] that write-update protocols have no coherence miss penalty.

- *F. Dahlgren and P. Stenström are with the Department of Computer Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. E-mail: {dahlgren, pers}@ce.chalmers.se.*
- *M. Dubois is with the Department of Electrical-Engineering Systems, University of Southern California, Los Angeles, CA 90089-2562. E-mail: dubois@paris.usc.edu.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106849.

1. Consistent with the literature, we use the word *latency* to designate the time taken by an access and the word *penalty* to designate the time that the processor is blocked on each access. The words *penalty* and *stall time* are used interchangeably throughout the paper.

The major drawback of these protocols is their large write memory traffic. *Competitive-update* protocols strike a compromise by mixing updates and invalidations. The idea is simple. Instead of invalidating a block copy at the first write by another processor, the block is at first updated; if the local processor does not access the block after a few updates, then the copy is invalidated. This scheme works very well under relaxed memory consistency models [12]. To further reduce the write traffic, a small *write cache* [7] can be used to buffer writes to the same block and to combine them before they are issued.

Because these three extensions to a basic directory-based write-invalidate protocol address different components of the read and write penalties, their gains may add up when they are combined. One focus of this paper is to identify combinations affording maximum performance gains within an architectural framework consisting of a directory-based cache-coherent NUMA architecture [19] based on single-issue, statically scheduled CPUs. We find that the combined techniques provide *additive gains* in many cases. For example, the performance improvement of adaptive prefetching combined with the competitive-update mechanism is often the sum of their individual improvements and yields, for some applications, a speedup of nearly a factor of two under release consistency. These gains are obtained by simple modifications to the *lockup-free caches* [8] and to the cache coherence protocol. A preliminary version of this paper is published in [6] and, in this paper, we extend that work by a closer analysis of some architectural parameters, such as cache and buffer sizes.

In order to compare implementations carefully, we start in Section 2 with the detailed design of our baseline architecture. In Section 3, we review the implementation details of the three protocol extensions as well as their expected performance gains in isolation. Based on detailed architectural simulation models and five benchmark programs from the SPLASH suite [18] introduced in Section 4, we evaluate the combined performance gains of the protocol extensions in Section 5. Finally, we compare the implications of the performance and implementation evaluations with work done by others in Section 6, before we conclude the paper in Section 7.

2 Architectural Framework and Baseline Architecture

None of the three protocol extensions we consider sets any specific requirement on the processor architecture. While we assume single-issue, blocking-load processors in our evaluation, the techniques also apply to multiple-issue processors with nonblocking loads. Fig. 1 shows the overall organization of each processing node and of the system. The environment of each processor is comprising a first-level cache (*FLC*), a second-level cache (*SLC*), and a first- and second-level write buffers (*FLWB* and *SLWB*). The processor and its caches are connected to the network interface and to a shared memory module by a local bus. The *FLC* is a direct-mapped, write-through cache with no allocation of blocks on write misses and is blocking on read misses. If the block is valid in the *FLC*, it is updated with the new value and the write request is buffered in the *FLWB* in FIFO order. Otherwise, the write request is buffered in the *FLWB* without any block frame allocation in the *FLC*. Read miss requests issued by the *FLC* are also buffered in FIFO order in the *FLWB*. The *SLC* is a direct-mapped write-back cache and maintains inclusion so that all blocks valid in the *FLC* are also valid in the *SLC*. Each write request give rise to a tag-check in the *SLC* before the block is allowed to be updated with the new value. If a write request has updated a valid block in the *FLC*, but that block becomes invalidated in both the *SLC* and *FLC* while the write request is still in the *FLWB*, that write request will be handled like a write-miss in the *FLC*, preventing the loss of the updated value.

The *SLC* and the *SLWB* designs incorporate most of the mechanisms to support each protocol extension. Unlike the *FLC*, which has to respond to all processor accesses and must be fast and simple, the *SLC* can afford sophisticated mechanisms for latency tolerance and reduction. The *SLC* is *lockup-free* [8] and buffers all pending requests (e.g., prefetches, updates, or invalidations) in a second-level write buffer (*SLWB*). This feature is critical in order to take advantage of relaxed memory consistency models [8] such as *release consistency* [10]. The protocol extensions require some extra control mechanisms in the lockup-free *SLC*s as well as some modifications to the system-level cache coherence protocol.

At the system level, the baseline architecture (henceforth referred to as *BASIC*) implements a write-invalidate protocol with a full-map directory similar to Censier and Feautrier's [2]. A presence-flag vector associated with each memory block points to the processor nodes with a copy in their cache. An *SLC* read miss sends a read miss message to the *home* memory module (the node at which the physical memory page containing the block is allocated). If the home memory is local and if the block is clean (unmodified), the miss is serviced locally. Otherwise, the miss is serviced either in two or in four node-to-node transfers depending on whether the block is dirty (modified) in some other cache. A write access to a shared or invalid copy in the *SLC* sends an ownership request to the home node; in response, the home node sends invalidations to all nodes with a copy, waits for acknowledgments from these nodes, and, finally, sends an ownership acknowledgment to the requesting node plus the copy of the block if needed.

Three state bits encode two stable and three transient states for each memory block. The stable states are CLEAN (the memory copy is valid) and MODIFIED (exactly one cache keeps the exclusive copy of the memory block). While the home node is waiting for the completion of a coherence action—e.g., the invalidation of copies—the memory block is in a transient state. If a read miss or ownership request reaches the memory while the block is in a transient state, it is rejected and must be retried.

There are three stable cache states: INVALID, SHARED, or DIRTY. No transient state is needed in cache because all pending accesses are kept in the *SLWB* of the requesting node until they are completed. For example, if a write is issued to a cache block in state SHARED, the cached copy is updated and an ownership request is buffered in the *SLWB*. Thus, the *SLC* can continue servicing local accesses for as long as there is space in the *SLWB*. Synchronizations such

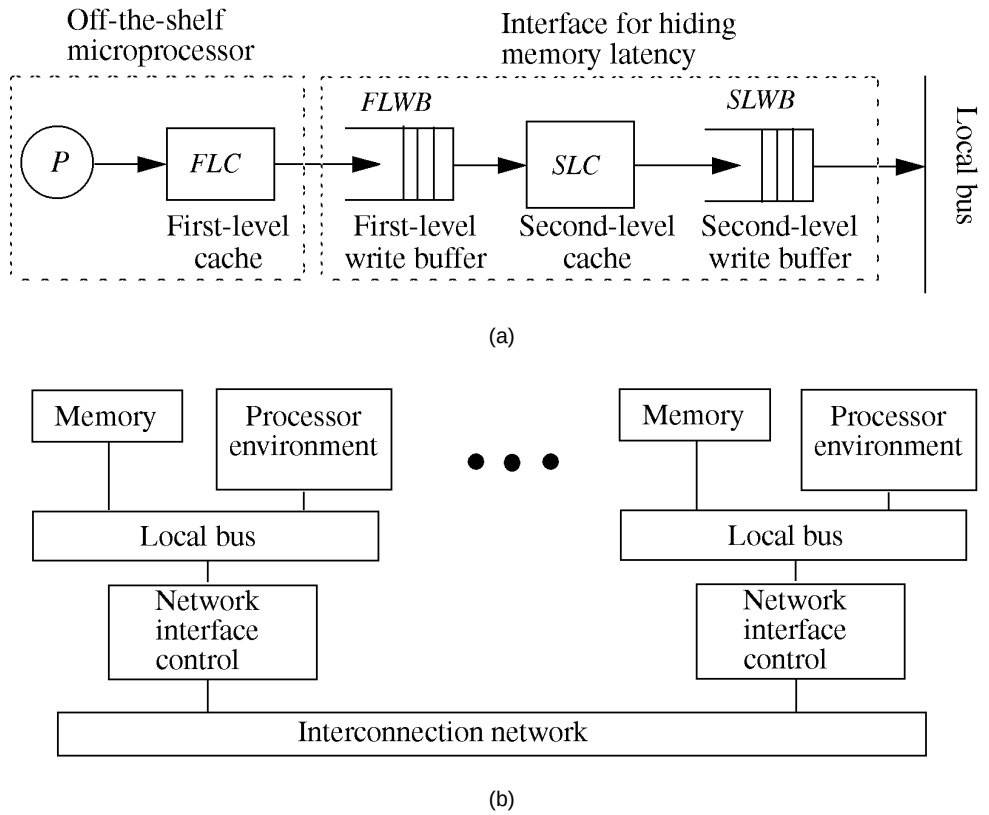


Fig. 1. The processor environment and the simulated architecture. (a) The processor environment. (b) The simulated architecture.

as *acquires* and *releases* bypass the *FLC* and are inserted in the *SLWB* with other memory requests. However, under release consistency, previously issued ownership requests must be completed before a release can be issued.

The hardware support for cache coherence in *BASIC* is limited to two bits per cache block and N + 3 bits per memory block for N nodes. Under sequential consistency, the read and write penalties are significant for large latencies. Under release consistency, multiple write requests can be overlapped with each other and with local computation to a point where the write penalty is completely eliminated [10]. This overlap is achieved through the lockup-free mechanism implemented in the *SLC* controller working in conjunction with the *SLWB*. The protocol extensions of the next section can reduce the read and the write penalties by simple modifications to the *SLC* and to the system-level cache coherence protocol.

3 Simple Cache Coherence Protocol Extensions

We now cover in some details the protocol extensions, including the hardware support needed beyond *BASIC*, and their effects on memory access penalties in Sections 3.1 to 3.3. We then discuss how the protocol extensions can be combined to further cut the penalties in Section 3.4. In the rest of this paper, we use the following classification of read misses. A miss is classified as a *cold miss* if the requested block has never been referenced by the processor. If the block has been referenced by the processor but has been written to by another processor, the miss is classified as a *coherence miss*. All other misses are referred to as *replacement misses* since they are caused by replacements from the cache because of its limited size and associativity.

3.1 Adaptive Sequential Prefetching

Nonbinding prefetching [13] cuts the read penalty by bringing into cache the blocks which will be referenced in the future and are not present in cache. The value returned by the prefetch is not bound because the prefetched block remains subject to invalidations and updates by the cache coherence mechanism. Software prefetching relies on the compiler or user to insert prefetch instructions statically into the code [16], whereas hardware schemes dynamically detect patterns in past and present data accesses to predict future accesses. Nonbinding prefetching is applicable to systems under any memory consistency model, including sequential consistency.

A nonbinding hardware-based prefetching scheme called *adaptive sequential prefetching* was proposed and evaluated experimentally in [4]. When a reference misses in the *SLC*, a miss request is sent to memory and the *K* consecutive blocks directly following the missing block in the address space are accessed in the cache to force a miss if needed. A Prefetch Counter in the *SLC* controller generates the addresses for these *K* blocks and a prefetch is inserted in the *SLWB* for each block missing in the cache unless a request for the block is already pending. (*K* is called the *degree of prefetching*.) The prefetches are issued one at a time, are pipelined in the memory system, and can be overlapped with the original read miss in the memory system.

TABLE 1
HARDWARE NEEDED TO SUPPORT *BASIC* AND THE HARDWARE OVERHEAD NEEDED BY EACH EXTENSION

	BASIC	*P* (Prefetch)	*M* (Migratory Optimization)	*CW* (Competitive-Update and Write Caches)
State bits per *SLC* line	2 bits (3 states)	2 bits (Prefetch and ZeroBit)	1 state	1 counter (Typically 1 bit)
Additional mechanisms per cache	None	3 counters (4 bits)	None	Direct-mapped write cache (Typically 4 blocks)
SLWB features	Number of entries is dictated by the consistency model	Outstanding prefetch requests are buffered	No extra entries (less entries than *BASIC* expected)	Each entry holds a block, but few entries suffice
State bits per memory line	3 state bits (for 2 stable states and 3 transient states) plus N presence bits	No extra state	1 state bit plus a pointer (log_2 *N* bits for *N* caches)	No extra state

This prefetching scheme exploits the spatial locality across cache blocks. Although one might believe that a block *K* times larger could have the same effect, it was shown in [4] that false sharing effects may nullify or even reverse the benefits of a bigger block size. Moreover, the need to adjust the degree of prefetching dynamically to variations in the spatial locality was demonstrated in [4]. If *K* remains fixed throughout the execution of a program, sequential prefetching is only marginally effective.

Conceptually, the degree of prefetching is controlled by counting the number of useful prefetches, i.e., the fraction of prefetched blocks which are later referenced by the processor. This fraction is compared to preset thresholds: If it is higher than the high mark, the degree of prefetching is increased, and if it is lower than the low mark, the degree of prefetching is decreased. Prefetching across page boundaries is not supported, i.e., a read miss on one page will never generate a prefetching for a block in another page, as it could potentially generate useless page faults. More details of this scheme are given in [4].

The adaptive prefetching scheme improves performance by cutting the number of cache misses. From the simulation of six benchmark programs, the scheme was shown to cut the number of cold and replacement misses. Contrary to common belief, the cold miss rate does not necessarily decline with time and may impact the overall performance of an application. This is true in general for direct (i.e., noniterative) solution methods in linear algebra, exemplified by LU and Cholesky used in Section 5. In these applications, the cold miss rate remains high during the whole execution. A significant reduction in the replacement miss rate for most of the applications was also reported in [4], which implies that the spatial locality of data references in these parallel applications is very high. Coherence misses are also cut in Cholesky and Water because of spatial locality in the true-sharing miss component. However, the false-sharing miss component is unchanged.

The hardware needed to extend *BASIC* with the adaptive prefetching scheme is three modulo-16 counters per cache and two bits per cache line. This overhead is recorded in column *P* in Table 1.

3.2 Migratory Sharing Optimization

While the adaptive sequential prefetching scheme cuts the read penalty, it does not improve the write penalty, which can be large under sequential consistency. In write-invalidate protocols, the write penalty comes from ownership requests to shared or invalid blocks. For memory blocks exhibiting *migratory sharing*, a sharing pattern in which a block is read and modified by different processors in turns, this penalty is particularly large because different processors in turn trigger a read miss followed by an ownership request. The miss is serviced by the cache attached to the last processor to access the block. The ownership request propagates a single invalidation to the same cache and could be avoided if the invalidation was done at the time the read miss is serviced. The protocol extension avoiding this invalidation is called the *migratory sharing optimization*. This optimization is aimed at a particular application behavior which is quite common and results from accesses to data in critical sections or in read/write sequences on shared variables occurring in statements such as "x := x + 1." (In the case of MP3D, migratory sharing is attributable to the latter.)

Both Cox and Fowler [3] and Stenström et al. [20] have indicated ways to extend a write-invalidate protocol with the migratory optimization. The detection is done at the home node, which sees read misses as well as ownership requests. A block is deemed migratory if the home node has detected a read/write sequence by one processor followed by a read/write sequence by another processor (see [3], [20] for more information about this). Stenström et al. [20] evaluate the performance advantage of the migratory optimization. For three applications (MP3D, Cholesky, and Water), the number of ownership requests are cut by between 69 percent and 96 percent and the execution time is reduced by as much as 35 percent (MP3D) under sequential consistency because of the lower write penalty. Performance can also be improved under release consistency because the migratory optimization cuts the write traffic as well, and the ensuing reduction of contention cuts the read and the synchronization penalties. This is critical for applications with higher bandwidth requirements than the network can sustain. We will demonstrate these effects in Section 5.

Compared to *BASIC*, the hardware overhead of the migratory optimization is an extra bit per memory line encoding two memory states (MIG-CLEAN and MIG-MODIFIED), a pointer of size $log_2 N$ bits (for N caches) per memory line, and an extra state (MIGRATING) per cache line [20]. This extra cache state is needed to disable the migratory optimization when the access pattern to a block deemed migratory changes to another form of sharing such as read-only sharing. These overheads appear in Table 1 in column *M*.

3.3 Competitive-Update Mechanisms

The above techniques are fairly inefficient at reducing the penalties associated with coherence misses. However, it is well known that a write-update protocol completely eliminates them. The cost is the increased number of update messages to shared blocks. Under a *relaxed memory consistency model* in conjunction with a sufficiently large *SLWB*, the latency of these updates can be hidden, as was shown in [10], but the memory traffic may offset the gains because of memory conflicts.

Competitive-update protocols, i.e., update-based protocols which invalidate a copy if the local processor does not access it sufficiently often, have been the subject of previous evaluations [12] in the context of an architecture similar to *BASIC*. The hardware cost is limited to a counter associated with each cache line. When a block is loaded into the cache, or when the block is accessed, the counter is preset to a value called the *competitive threshold*. The counter is decremented every time the block is updated by another processor. When the counter reaches zero, the block is invalidated locally. Thus, if a number of global updates equal to the competitive threshold reach the cache with no intervening local access, the block is invalidated locally and the propagation of updates to that cache is stopped.

Two factors contribute to the good performance of a competitive-update protocol [7], [12]:

1) The coherence miss rate is reduced and
2) The latencies of the remaining coherence misses are shorter because the likelihood of finding a clean copy at memory is higher under a competitive-update protocol than under a write-invalidate protocol.

In [12], the competitive-update protocol briefly described here was shown to outperform write-invalidate for all applications, although the performance improvement for applications with migratory sharing was limited. In order for these applications to benefit from write-update, the competitive threshold needs to be large, which in turn tends to increase the write traffic and thus reduce the benefits. A competitive threshold of four was recommended in [12] as a good compromise.

This simple competitive update mechanism is even more effective in conjunction with write caches [7]. A write cache is a small cache which allocates blocks on write requests only. Because consecutive writes to the same word are combined in the write cache before being issued, the write traffic is reduced. This combining is only possible under a relaxed memory consistency model which allows for delays in the propagation of writes until a synchronization point. For example, under release consistency, the propagation of updates to a block in the write cache can wait until the block is replaced or until the release of a lock.

In *BASIC*, the write cache must be attached to the *SLC* so that the write cache can be accessed at the same time as the *SLC*. For read accesses hitting in the *SLC* and for write accesses to dirty blocks in the *SLC*, no write cache action is taken. On an *SLC* read miss hitting in the write cache, the data is returned to the processor. However, if the read request also misses in the write cache, the block is fetched into the *SLC* from memory as usual. A block fetch is not triggered by a write miss in the *SLC*. Instead, a write to a shared or invalid block in the *SLC* allocates a block frame in the write cache; the new value is stored in the write cache and is not propagated to other caches. Subsequent writes to the same block are combined in the write cache until the block is eventually written to the home node at the next release or at the replacement of the block in the write cache. To keep track of the modified words in a block of the write cache, a dirty / valid bit is associated with each word. These bits are also used to selectively send the modified words to the home node, further reducing traffic. However, a single header is needed for such partial block transfers, thus reducing traffic. The write cache blocks that are written to memory on a replacement or at a synchronization point are temporarily buffered in the *SLWB*.

Detailed performance evaluations reported in [7] show that a direct-mapped write cache with only four blocks is very effective at combining writes to the same block. Moreover, after a synchronization point or after a block is victimized, the probability that the block will be accessed by a different processor is very high. Therefore, a competitive update protocol with write caches and a threshold of one will in general exhibit less network traffic because of combined writes and lower coherence miss penalty than a competitive-update protocol using a threshold of four and no write caches.

To conclude, the extension of *BASIC* with a competitive-update mechanism with write caches requires a modulo-2 counter per cache line (one bit). However, the *SLWB* is somewhat more complex because each entry now contains a block. The overheads associated with this technique are summarized in Table 1 in the column denoted *CW*.

3.4 Combining the Techniques

We now consider the implications of combining the protocol extensions. To simplify, we refer to *BASIC* plus adaptive sequential prefetching as *P*, to *BASIC* plus the migratory optimization as *M*, and to *BASIC* plus the competitive-update mechanism and write caches as *CW*. For example, the mechanisms needed to implement *P* can be derived from Table 1 by the hardware mechanisms under *BASIC* plus the hardware mechanisms under *P*.

The combination of *P* with *CW* (*P* + *CW*) is expected to remove almost all misses since *P* is aimed at cold and replacement misses and *CW* is aimed at coherence misses. This will result in a very small read penalty and high performance gains, as we will show in Section 5. These techniques are trivially combined; no new hardware resource and no modification to the cache coherence protocol is needed. Thus, the hardware mechanisms needed appear in Table 1 under *BASIC* plus *P* plus *CW*.

Whereas *P* mainly reduces the read penalty due to cold and replacement misses, *M* cuts the write penalty and traffic associated with migratory blocks. Thus, a combination of *P* and *M*, referred to as *P* + *M*, is expected to reduce the read as well as the write penalties for migratory blocks. Since prefetch requests to MIGRATORY blocks retrieve exclusive copies (because they are seen as read misses by the home node), prefetching in *P* + *M* is equivalent to a hardware-based read-exclusive prefetching scheme [16]. From an implementation viewpoint, this combination does not need any additional hardware resource besides those listed in Table 1 for *P* and *M*.

The combination of *CW* and *M* (*CW* + *M*) has the following implications. Since *CW* is only applicable to implementations under relaxed memory consistency models, there is no write stall time. Since optimizations for migratory sharing, *M*, aim at reducing the number of global write requests, the potential gains of adding *M* to *CW* must come from reduced traffic and less network contention.

With the competitive-update mechanism, the home node cannot detect that two consecutive nonoverlapping read/write sequences by distinct processors are migratory because it sees only updates and not local reads. To make sure that read/write sequences are nonoverlapping, the following heuristic is used when the home node receives an update request. If the number of cached copies is greater than one and the update request comes from another processor than the last update request, the block is *potentially* regarded as migratory. To deem it migratory, the home node interrogates all caches that have copies. Upon receipt of this request, each cache responds in one of two ways. If the block has not been modified locally at all, or, if the block has been read but not modified since the last update from the home node, the block is *not* deemed migratory. Otherwise, the cache gives up its copy and acknowledges home. For the block to be deemed migratory, all caches must give up their copies. An extra bit is needed in the cache to keep track of whether or not a block has been locally modified.

Finally, combining all three techniques, i.e., *P* + *CW* + *M*, does not require any further modifications beyond the ones already described in this section.

4 Simulation Methodology and Benchmark Programs

We have developed simulation models of *BASIC* and all its protocol extensions presented in the previous section. The simulation platform is the CacheMire Test Bench [1], a program-driven functional simulator of multiple SPARC V8 processors. It consists of two parts:

1) a functional simulator and
2) an architectural simulator.

The SPARC processors in the functional simulator issue memory references and the architectural simulator delays the simulated processors according to its timing model. Consequently, the same interleaving of memory references is maintained as in the target system we model. To reduce the simulation time, we simulate all instructions and private data references as if they always hit in the *FLC*.

TABLE 2
Fixed Architectural Parameters

Parameter	Value (1 pclock = 10ns)
Number of processors	16
First-level cache (*FLC*) size	4 Kbytes
Block size (*FLC* + *SLC*)	32 bytes
Read from *FLC*	1 pclock
Read from *SLC*	6 pclocks
Read from local memory	30 pclocks

We have carefully selected some architectural parameters that remain the same in all simulations. These parameters appear in Table 2. The timing assumptions are based on a processor clock (or pclock) rate of 100 MHz. We assume single-issue, statically scheduled processors. The *FLC* has the same cycle time as the processor with an *FLC* fill time of three cycles. The *SLC* is built from static RAMs with a cycle time of 30 ns. We assume a fully interleaved memory with an access time of 90 ns, and a 256-bit wide local split-transaction bus clocked at 33 MHz.

The variable parameters include the sizes of the *FLWB*, of the *SLWB*, and of the *SLC*. By default, we assume a contention-free uniform access time network with a node-to-node latency of 54 pclocks. Contention is accurately modeled in each node. To specifically study the impact of network contention, we consider, in some cases, a detailed model of a mesh with wormhole-routing. The design considerations for the mesh are discussed in detail in Section 5. Finally, synchronization is based on a queue-based lock mechanism at memory similar to the one implemented in DASH, with a single lock variable per memory block. In addition, pages (4 Kbytes) are allocated across nodes in a round-robin fashion based on the least significant bits of the virtual page number.

We use five benchmark programs to drive our simulation models. These are summarized in Table 3. Three of them are taken from the SPLASH suite (MP3D, Water, and Cholesky) [18]. The other two applications (LU and Ocean) have been provided to us from Stanford University. They are all written in C using the ANL macros to express parallelism and are compiled by gcc (version 2.1). For all measurements, we gather statistics during the parallel sections only according to the recommendations in the SPLASH report [18].

5 Experimental Results

In Section 5.1, we compare the performance gains obtained by each protocol extension in isolation and by each of the four combinations, assuming infinite caches, sufficient network bandwidth, and local buffering under release consistency. In Sections 5.2 through 5.5, we impose various architectural constraints: sequential consistency in Section 5.2, network bandwidth limitations in Section 5.3, buffer size limitations in Section 5.4, and finite cache sizes in Section 5.5. Finally, we contrast the hardware cost of each extension with its performance gain in Section 5.6.

5.1 Individual and Combined Performance Gains

In this section, the individual and combined performance gains of all protocol extensions are compared to the

TABLE 3
BENCHMARK PROGRAMS

Benchmark	Description	Data Sets
MP3D	3D particle-based wind-tunnel simulator	10 K parts, 10 time steps
Water	N-body water molecular dynamics simulation	288 molecules, 4 time steps
Cholesky	Cholesky factorization of a sparse matrix	matrix bcsstk14
LU	LU-decomposition of a dense matrix	200 × 200 matrix
Ocean	Ocean basin simulator	128 × 128 grid, tolerance 10^{-7}

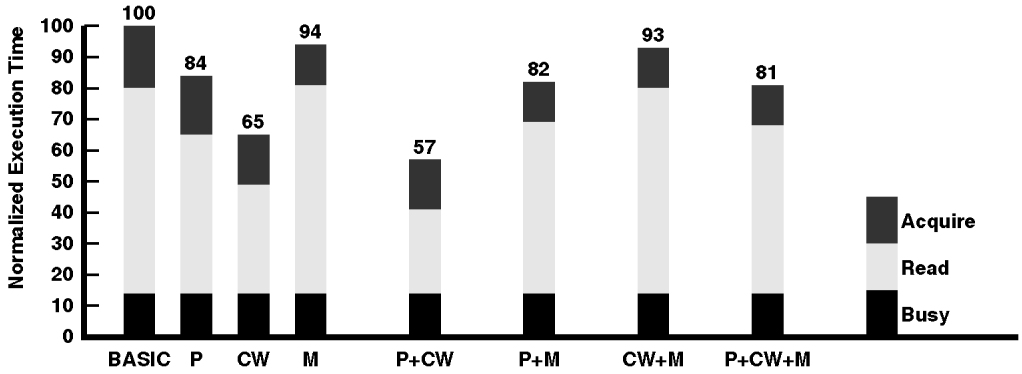


Fig. 2. Execution times of MP3D relative to *BASIC* under release consistency.

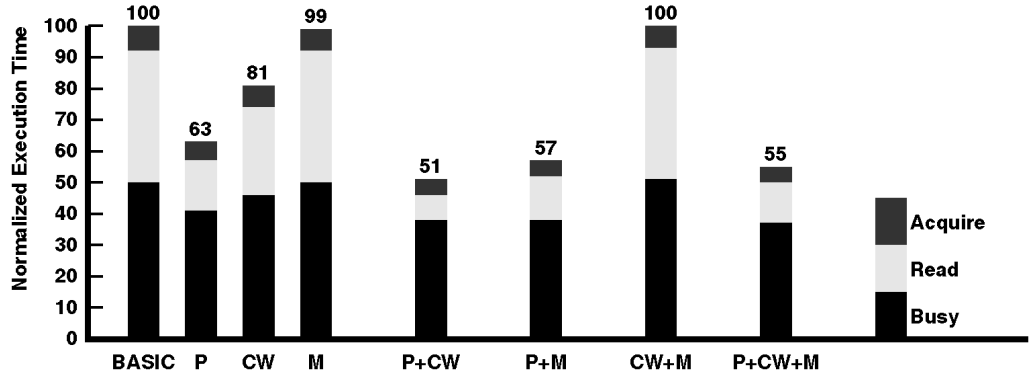


Fig. 3. Execution times of Cholesky relative to *BASIC* under release consistency.

performance of *BASIC* under release consistency.[2] *BASIC* exploits the write latency tolerance afforded by release consistency with a 16 entry deep *SLWB* (as many as 16 writes can be pending in each node) and an eight entry deep *FLWB*. Henceforth, if not stated explicitly otherwise, *BASIC* will refer to this implementation.

Since latency tolerance techniques in general require more bandwidth and since our purpose in this section is to understand the gains, given enough network bandwidth, we simulate a network with infinite bandwidth, although we carefully model contention in each node. Later, in Section 5.3, we will consider the impact of network contention in detail. To concentrate on cold and coherence miss penalties, we also limit ourselves to infinite second-level caches in this section.

2. Our release consistency implementation allows an acquire to bypass a previous release request provided that they are directed to different blocks. This is in accordance with Gharachorloo's *RCpc* model [10].

Figs. 2, 3, 4, 5, and 6 show the execution times for each benchmark program relative to *BASIC*. The execution time for each protocol extension is further decomposed into the fraction of busy time (the time the processor is executing instructions), read stall time (the time the processor is waiting for read requests to complete), and acquire stall time (the time spent waiting for an acquire to complete). Since the write latency is completely hidden, it does not contribute to the execution time for any of the protocols.

Let us focus on the gains of each individual protocol extension first. The three bars to the right of *BASIC* correspond to the relative execution times of *BASIC* with adaptive sequential prefetching (*P*), with the competitive-update mechanism (*CW*), and with the migratory optimization (*M*) for each application. As can be seen, *P* and *CW* are the most successful protocols. *P* manages to cut the read stall time because of cache miss reductions for all applications except Ocean. To see this, we show the cold and coherence miss rates for *P* and for *BASIC* in Table 4. For example, *P* manages

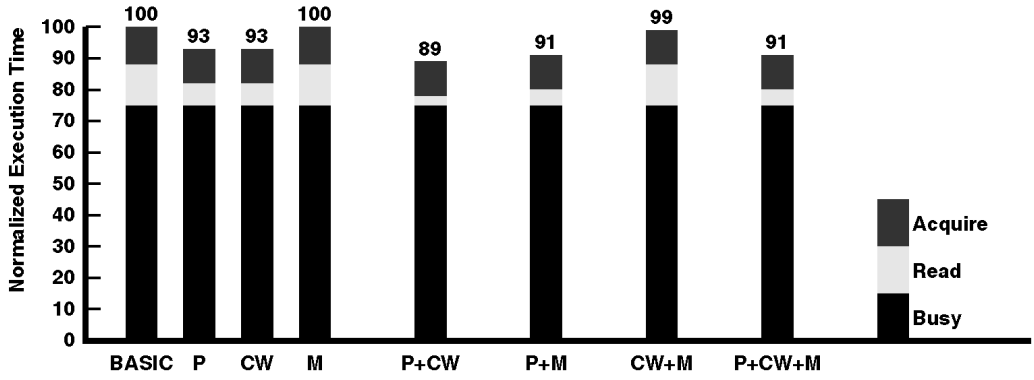


Fig. 4. Execution times of Water relative to *BASIC* under release consistency.

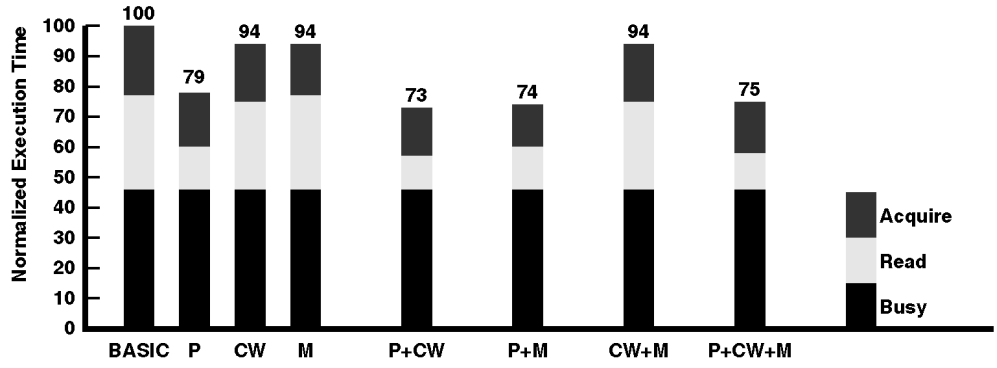


Fig. 5. Execution times of LU relative to *BASIC* under release consistency.

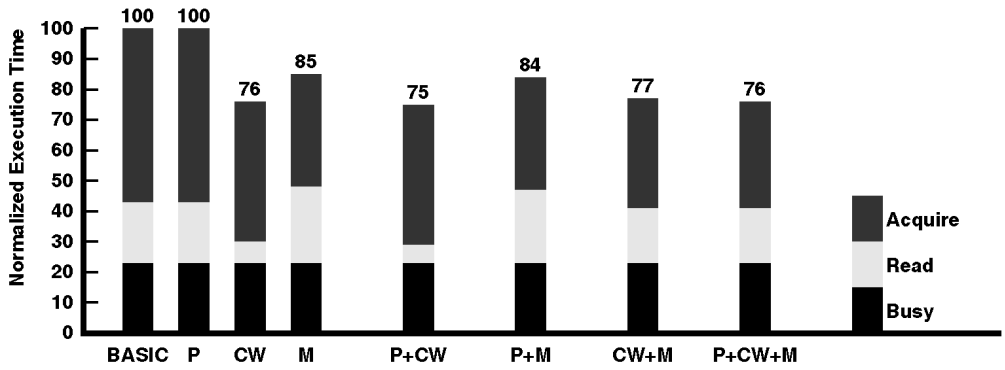


Fig. 6. Execution times of Ocean relative to *BASIC* under release consistency.

TABLE 4
COLD AND COHERENCE MISS RATE COMPONENTS FOR *BASIC*, *P*, *CW*, AND *P + CW*

	BASIC		*P*		*CW*		*P + CW*	
Application	Cold	Coherence	Cold	Coherence	Cold	Coherence	Cold	Coherence
MP3D	1.3%	9.0%	0.7%	7.5%	1.3%	8.0%	0.6%	6.3%
Cholesky	0.90%	0.30%	0.19%	0.09%	0.91%	0.26%	0.18%	0.07%
Water	0.04%	0.72%	0.01%	0.24%	0.04%	0.63%	0.01%	0.22%
LU	0.86%	0.05%	**0.22%**	0.06%	0.86%	**0.01%**	**0.22%**	**0.01%**
Ocean	0.02%	0.72%	0.01%	0.69%	0.02%	0.20%	0.01%	0.18%

to cut the cold miss rate from 0.86 percent to 0.22 percent in LU, and from 0.90 percent to 0.19 percent in Cholesky. Moreover, *P* also removes some of the coherence misses in MP3D, Cholesky, and Water.

The protocol with the competitive-update mechanism (*CW*) uses a competitive threshold of one and a fully associative[3]

3. In [7], it is shown that the organization of the write cache is not critical; a direct-mapped write cache would perform nearly as well.

write cache of four blocks with a FIFO replacement policy. *CW* manages to cut the read stall time substantially for all applications but LU. These gains are entirely attributed to the reduction of the coherence miss penalty. Table 4 confirms this expectation. While the coherence miss rates are cut by *CW* for all applications, the cold miss rates are almost unaffected. In the case of MP3D, *CW* only manages to reduce the coherence miss rate from 9 percent to 8 percent (see Table 4). Nonetheless, the read penalty reduction is significant and is essentially due to the shorter latency of the remaining coherence misses, which are now serviced mostly by the home node because the memory copy is more often clean than in *BASIC*. To confirm this, we measured the average time to handle a read miss for MP3D and found that it is 41 percent shorter under *CW* than under *BASIC*.

Finally, since the write latency is completely hidden under release consistency, the contribution of M is indirect and comes from the reduced write traffic and the ensuing reduction of the number of pending writes. The write traffic reduction has virtually no impact on the read stall time because we model an infinite bandwidth network; rather, it helps cut the acquire stall time, as is clearly visible in the case of MP3D (Fig. 2).

Moving on to the combined performance gains, we see that combining *P* and *CW* (*P* + *CW*) leads to additional reductions of the read stall times for all applications as compared to *P* or *CW* alone. For the sake of illustration, consider the case of LU in Fig. 5. The combined gain of *P* + *CW* is the sum of the gains of *P* and *CW* alone. The reason for this can be seen from Table 4, which shows that the cold miss rates for *P* and *P* + *CW* are the same and that the coherence miss rates of *CW* and *P* + *CW* are also the same. (These values are entered in boldface in Table 4.) The same observations are applicable to the other programs. The combined gains of *P* and *CW* lead to a performance improvement of nearly a factor of two (with respect to *BASIC*) for MP3D and Cholesky, and to a reduction of between 60 percent and 81 percent of the read stall time for all applications.

Because *M* can only cut the acquire stall time under release consistency, *P* + *M* performs only slightly better than *P* alone. Combining *M* with *CW*, we note that the gains of *CW* are wiped out for all applications exhibiting a significant degree of migratory sharing (MP3D, Cholesky, and Water). Indeed, *CW* helps these applications by keeping the memory copy clean, whereas *M* promotes the existence of a single copy. Thus, *CW* + *M* is not a useful combination in an architecture implementing release consistency and with enough network bandwidth. The combination of all three techniques (*P* + *CW* + *M*) almost has the same performance as *P* + *M* for all applications and, therefore, is not a useful combination.

In summary, under release consistency and with enough network bandwidth to accommodate the bandwidth requirements, *P* + *CW* provides a significant performance improvement because of the additive effects of miss penalty reduction. By contrast, *M* and its combinations with *P* or *CW* are not useful in this case because there is no write penalty under release consistency.

5.2 Combined Gains under Sequential Consistency

An advantage of release consistency is the complete elimination of the write penalty. Unfortunately, relaxed consistency models in general tend to add complexity to software developments. Since sequential consistency offers a more intuitive programming model, techniques to reduce the read stall time, *as well as* the write stall time under sequential consistency, are therefore very useful. In this section, we evaluate the performance gains of *P* and *M* under sequential consistency; we omit *CW* because it is not feasible under sequential consistency. We implement sequential consistency in all designs in this section by stalling the processor for each issued shared memory reference until it is globally performed. Therefore, a single entry suffices in the *FLWB* for *BASIC*, *M*, and *P*. Moreover, under *BASIC* and *M*, only a single entry in the *SLWB* is needed in contrast to *P* which still needs to keep track of pending prefetch requests. For the purpose of comparing the different systems with respect to the *SLC* and the *SLWB* designs, we refer to the sequential consistency implementations of *BASIC* and *M* as *B-SC* and *M-SC*, respectively. Note that since *P* under sequential consistency still needs a *SLWB*, it does not differ in this respect from a release consistent implementation.

Figs. 7 and 8 show the execution times for *P*, *M-SC*, and *P* + *M* under sequential consistency for all applications. Each bar is decomposed into the same time components as in Section 5.1, with the addition of the write stall time (due to pending write requests) and of the release stall time (due to pending release requests).

The read stall time in *P* is quantitatively reduced by about the same amount (with respect to *B-SC*) as under release consistency. The write stall time is either the same or is slightly increased as compared to *B-SC*. This slight increase of the write stall time is a side effect of prefetching, which tends to increase the number of propagated invalidates because of a larger number of cached copies. In some cases, a prefetched copy may be invalidated before it is accessed by the local processor. However, this effect is small because the adaptive scheme adjusts the degree of prefetching according to the prefetching efficiency. The overall execution time reduction is at most 26 percent for Cholesky as compared to *B-SC*.

Since *M-SC* is aimed at the write and acquire stall times for migratory blocks, the execution time is significantly shorter for MP3D, Cholesky, and Water. Although Ocean does not exhibit any significant migratory sharing, the write and the acquire stall times are nonetheless reduced. We speculate that false sharing interactions cause blocks to become migratory at times. Although some ownership requests are removed and, thus, the write and acquire stall times are cut, the read miss rate and the read stall time are slightly increased. Overall, the execution time reduction in *M-SC* is at most 39 percent (MP3D) as compared to *B-SC*.

We have seen that *P* attacks the read stall time and that *M* attacks the write and the acquire stall times. Therefore, the combination of the two techniques might help reduce all three components. Fig. 7 confirms this expectation; *P* + *M* manages to improve performance significantly for MP3D, Cholesky, and Water, which are applications with significant migratory sharing. Again, we see an example of a

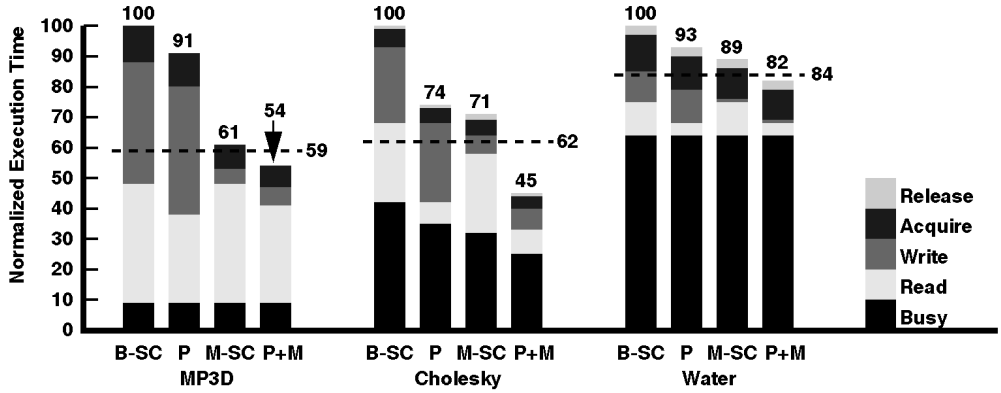


Fig. 7. Execution times of MP3D, LU, and Cholesky for *B-SC*, *P*, *M-SC*, and *P + M* under sequential consistency. The dashed line corresponds to *BASIC* under release consistency.

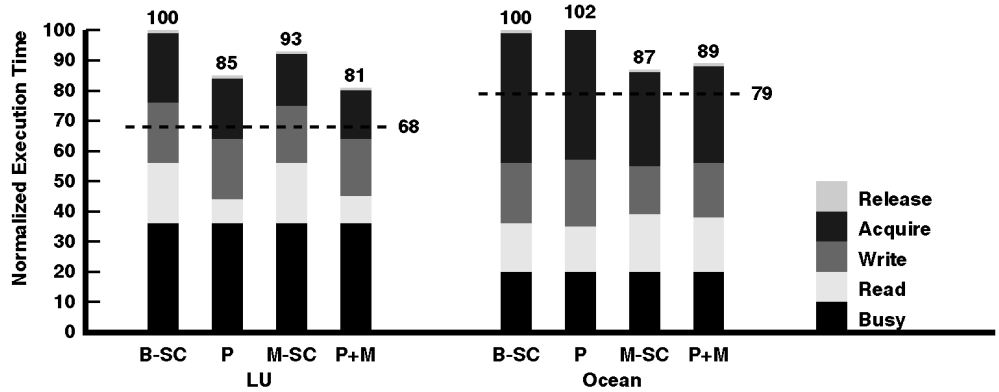


Fig. 8. Execution times of Water and Ocean for *B-SC*, *P*, *M-SC*, and *P + M* under sequential consistency. The dashed line corresponds to *BASIC* under release consistency.

combination where the gains are additive; the read stall times of *P* and *P + M* are almost the same, as are the write and the acquire stall times of *M-SC* and *P + M*. For Ocean, the slight increase in execution time caused by prefetching does also show up, together with the migratory optimization. For LU, there is a small load-imbalance which is improved by the faster execution in *P* as well as *M*, but the combination does not lead to further improvements. Therefore, the additional improvement of adding *M* to *P* is only 5 percent (81 percent relative to 85 percent), while it is 7 percent (93 percent relative to 100 percent) without *P*. The combined gains lead to an execution time reduction of 46 percent for MP3D and 55 percent for Cholesky. *P + M* is, to the best of our knowledge, the first *hardware-based read-exclusive prefetching scheme* reported in the literature.

There are two possible side effects of combining *M* with *P*. First, useless exclusive prefetches may lead to situations where migratory blocks currently under modification by one processor are exclusively prefetched by another cache. This type of occurrence would increase the read stall time as compared to *P* without *M*. Second, since *P* slightly increases the memory traffic, the write stall time of *P + M* might be slightly higher than that of *M*. However, Figs. 7 and 8 show that these effects are negligible.

In summary, under sequential consistency, *P* and *M* cut the read and write penalties, respectively, and their combination is equivalent to a very effective hardware-based read-exclusive prefetching scheme. Finally, to compare the execution times under sequential consistency and under release consistency, we show the execution time for *BASIC* (using an *SLWB* with 16 entries) in the diagrams of Figs. 7 and 8 (dashed lines). The combined *P + M* scheme under sequential consistency manages to outperform *BASIC* under release consistency for three out of the five applications.

5.3 Effect of Limited Network Bandwidth

In order to evaluate the effect of network contention on the behavior of individual and combined protocol extensions, we have run the same simulations with three different mesh networks assuming three link widths: 64, 32, and 16 bits. The meshes are wormhole-routed with two-phases (routing + transfer), meaning that each additional link requires two network cycles unless the message is affected by contention. Furthermore, the mesh is clocked at the same frequency as the processors (100 MHz). These network implementations are not overly aggressive in order to stress the impact of contention on the cache protocol extensions.

In this section, we will only concentrate on two combinations, *P + CW* and *P + M*, under release consistency because these combinations proved to be effective under infinite network bandwidth. Fig. 9 shows the execution times in system *P + CW* for all applications and for all three

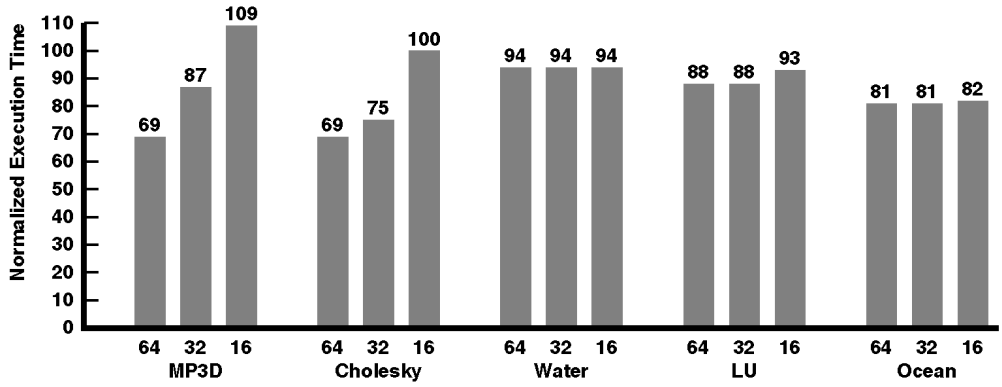


Fig. 9. Execution times for *P + CW* in systems with different mesh link widths normalized to *BASIC* with the same link width.

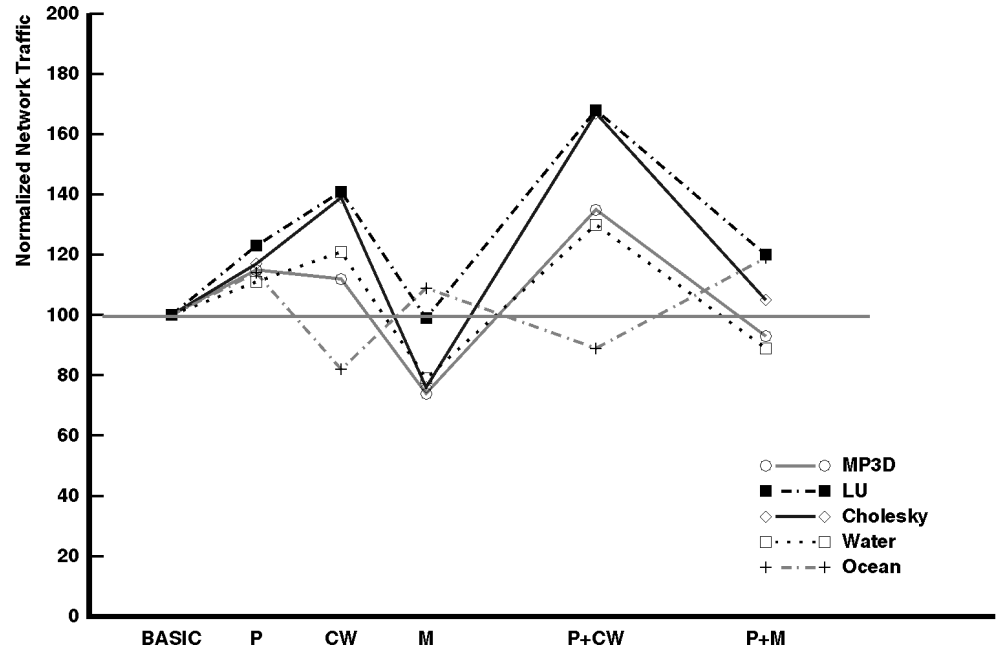


Fig. 10. Total amount of network traffic generated (in bytes) normalized to that of *BASIC*.

meshes. The execution times are normalized to that of *BASIC* for the same mesh and the same application. For example, in the case of a 64- bit mesh, the execution time of *P + CW* is only 69 percent of that of *BASIC* for MP3D (the leftmost bar). For 16-bit links, we see that the execution time for *P + CW* is 9 percent longer than for *BASIC*. This means that the relative advantage of *P + CW* over *BASIC* for MP3D is higher in systems with wider links.

In order to better comprehend the effect of network bandwidth on the execution time, we show in Fig. 10 the total amount of network traffic generated in different systems under release consistency. For MP3D the traffic is 36 percent higher in *P + CW* than in *BASIC*. For three applications (LU, Water, and Ocean), the differences between the relative execution times in systems with different link widths are very small. It appears that, whereas the difference between systems with 64 and 32-bit links is very small, the gap widens between systems with 32-bit and 16-bit links. This trend is also present in *P + CW* for Cholesky (Fig. 9): In systems with 32-bit and 64-bit links, the execution time is 25 percent and 31 percent shorter than in *BASIC* but these gains vanish in systems with 16-bit links. Fig. 10 reveals that the total amount of network traffic generated by Cholesky in *P + CW* is about 70 percent higher than in *BASIC*. This traffic increase explains why *P + CW* is more sensitive to network contention than *BASIC*. It appears however that 32-bit and 64-bit links are wide enough to keep network contention low whereas 16-bit links bring the network to saturation.

The same underlying trend is apparent in the case of the other applications, but different applications have different bandwidth requirements and, therefore, saturate the network for different link widths. For example, MP3D produces a lot of traffic and is already affected by higher traffic in systems with 32-bit links; on the other hand, the performance of Water is independent of the link width even down to 16-bit links. Overall, for all applications except MP3D, link widths of 32 and 64 bits yield about the same execution time in *P + CW* relative to *BASIC*, indicating that *P + CW* is an extremely competitive combination at reasonable network bandwidths.

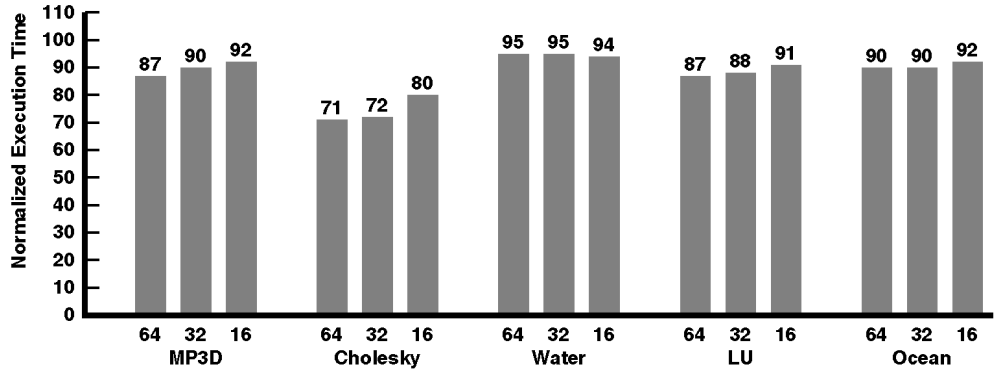


Fig. 11. Execution times for *P* + *M* in systems with different mesh link widths, normalized to *BASIC* with the same link width.

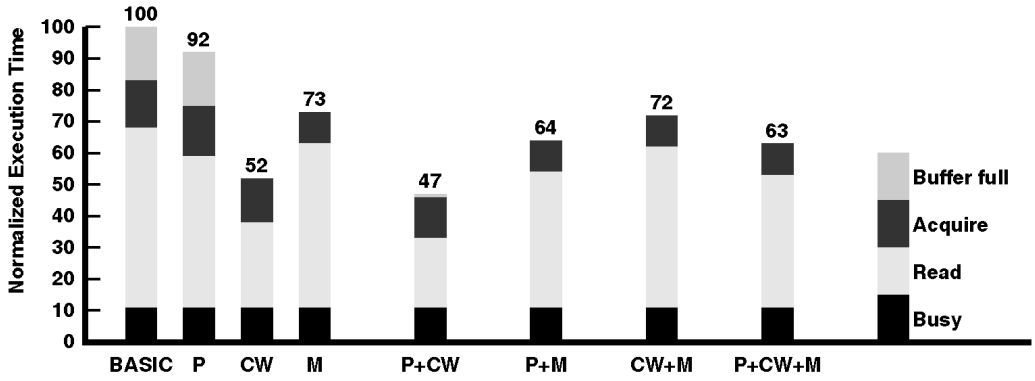


Fig. 12. Performance of MP3D with an *FLWB* and an *SLWB* of four entries each.

In contrast to *P* + *CW*, the difference in network traffic between *P* + *M* and *BASIC* is small for all applications, as can be observed from Fig. 10. As a direct result, the relative improvement of *P* + *M* over *BASIC* is not very sensitive to network contention. This is confirmed in Fig. 11, in which the sensitivity to the link width of the relative execution times is very small. Compared to *P* + *CW*, the gains of *P* + *M* are not affected as much by network contention.

We can compare the performance of *P* + *CW* and *P* + *M* from Figs. 9 and 11. Whereas *P* + *CW* provides the largest performance improvement given sufficient network bandwidth, *P* + *M* is better for three out of five applications for the most conservative network. Although *P* and *P* + *M* reduce the number of read misses to about the same extent, the increase in traffic due to prefetching is compensated by a smaller write traffic provided by the migratory optimization in *P* + *M*. As a result, *P* + *M* is shown to be a promising way of utilizing hardware prefetching when the network bandwidth is limited.

In summary, whereas the traffic generated by *P* + *CW* can reduce its gains for conservative network designs, *P* + *M* is much less sensitive to network contention because the bandwidth freed by the migratory optimization is available to the prefetch mechanism.

5.4 Effects of Limited Buffer Sizes

We have run the same simulations as in Section 5.1 but with smaller *FLWB* and *SLWB* (four entries in each). Due to space limitations, we only show the results for MP3D. The results for the other applications are similar.

In Fig. 12, we show the execution times relative to *BASIC*. The uppermost section of each bar represents the fraction of time the processors are stalled due to a full *FLWB*. *BASIC* now shows a significant buffer stall time (18 percent). Since the processors block on read requests (implying at most one pending read request), this stall time is due to write requests. Most reads are completed in the *FLC*, whereas all writes must be serviced by the *SLC* (the *FLC* is write-through). If the *SLWB* is too small, a sequence of write requests to invalid or shared cache blocks with no intervening read miss request could fill it. This is because writes to *SLC* blocks with pending ownership or write miss requests must be kept in the *SLWB*. When this buffer is full, the *SLC* is blocked, which eventually blocks the processor because of a backed-up *FLWB*.

We see in Fig. 12 that the execution time of *P* has about the same amount of buffer stall time as *BASIC*. One would expect that hardware prefetching could lead to longer buffer stall times since pending prefetches kept in the *SLWB* may fill it. However, since the processor is blocked while the prefetches are issued (because they are issued after a miss), the risk that the *FLWB* backs up is low. On the other hand, the smaller amount of free space in the *SLWB* limits the number of prefetches that can be issued, so the small size of the *SLWB* limits the potentials of reducing read misses by prefetching. However, for the applications studied, the cut in the read stall time in *P* (with respect to *BASIC*) is approximately the same with these smaller buffer sizes as with the larger buffers used in Section 5.1, indicating that the buffer stall time of *P* is attributable to writes.

The buffer stall times are negligible in both *CW* and *M*, indicating that these systems need smaller *SLWB*s than does *BASIC*. This observation is also true for any combination including *CW* or *M*. The gains from a reduced number of read misses and reduced read miss latencies in *P* + *CW* and *P* + *M* are still realized, even with small buffer sizes. Overall, except for the buffer stall times in *BASIC* and *P*, the simulation results are basically identical with those of Section 5.1.

5.5 Effects of Limited Cache Sizes

We have also run simulations of systems with a direct-mapped 16-Kbyte *SLC*. Overall, the qualitative conclusions are not affected by the cache size in significant ways. The read stall time is relatively higher due to a much larger number of misses (because of replacement misses). Adaptive sequential prefetching in general works just as well at reducing the number of replacement misses as the number of cold and true-sharing misses. In fact, for applications such as Ocean, where replacement misses to consecutive blocks occur in bursts, the impact of adaptive prefetching grows with smaller caches and adding prefetching improves the performance of the system with finite caches significantly. For MP3D, LU, Cholesky, and Water, adding prefetching has about the same effect for finite or for infinite caches. This is equally true for all combinations including adaptive sequential prefetching.

5.6 Cost-Effectiveness Analysis

To summarize our findings, we compare in this section the cost and effectiveness of each combination under various architectural constraints. Under release consistency, we found that *P* + *CW* can effectively cut the read penalty relative to *BASIC* by between 60 percent and 81 percent, given enough network bandwidth. Under limited network bandwidth, *P* + *M* is attractive because the migratory optimization reduces the traffic so that the network can accommodate the extra traffic generated by prefetching.

We have shown in Section 5.4 that an *SLWB* with only four entries is not sufficient to eliminate the buffer stall times in *BASIC* and *P*. Although a buffer size between four and 16 can remove some of the losses due to full buffers, we will base our cost analysis on the assumption that 16 entries are needed. In *BASIC*, each buffer entry contains the word address (four bytes), the command type (one byte), and the data to be written (four bytes). By contrast, each buffer entry in *CW* contain the block address, the command type, a copy of the whole block (32 bytes), along with a bit mask with a single bit per word (one byte). Table 5 shows the number of entries and sizes of the *FLWB* and *SLWB* in each system. The buffering requirements in *P* + *CW* are about the same as in *BASIC*, which implies that the additional complexity of *P* + *CW* is due to the write cache (only four blocks) and the prefetching mechanism. The buffering requirements for *P* + *M* are significantly lower than *BASIC*, and the overall hardware cost of *P* + *M* is thus smaller. Given the performance improvements indicated in this paper and the already significant complexity of *BASIC*, we conclude that the additional hardware complexity of *P* + *CW* as well as of *P* + *M* is marginal and cost-effective.

TABLE 5
BUFFERING REQUIREMENTS UNDER RELEASE CONSISTENCY

	FLWB #entries	*SLWB* #entries	*SLWB* #bytes
BASIC	8	16	144
P	8	16	144
CW	4	4	152
M	4	4	36
P + *CW*	4	4	152
P + *M*	4	4	36

TABLE 6
BUFFERING REQUIREMENTS UNDER SEQUENTIAL CONSISTENCY

	FLWB #entries	*SLWB* #entries	*SLWB* #bytes
B-SC	1	1	9
P + *M*	1	4	36

Table 6 shows the buffering requirements for *B-SC* (the sequential consistent implementation of *BASIC* in Section 5.2) and *P* + *M* under sequential consistency. A single entry in the *FLWB* and in the *SLWB* suffices in *B-SC*. Since the prefetch efficiency of *P* is the same with four as well as 16 entries, as shown in Section 5.4, *P* + *M* requires only four entries. Thus, *P* + *M* requires one *FLWB* and four *SLWB* entries, which is less than the requirements of *P* + *M* under release consistency. *P* + *M* needs more buffer space than *B-SC*, but significantly less buffering than *BASIC* (see Table 5). According to Table 1, the additional mechanisms to implement *P* and *M* are few. Therefore, we consider *P* + *M* to be a cost-effective optimization under sequential consistency.

It should be noted that one could consider the same *SLWB* organization under *BASIC* as under *CW*. However, this would necessitate the need of merging each write request to a pending block into the block frame of the *SLC* once ownership is granted. Moreover, the same buffering requirement as *CW* would still be needed. Conversely, one could also consider the same *SLWB* organization for *CW* as for *BASIC*. The disadvantage of this approach is the increased traffic because all modified words in a write cache block frame would then be propagated individually with individual headers.

6 Related Work and Discussion

Gupta et al. compared four latency tolerance and reduction techniques [13] consisting of coherent caches (with a write-invalidate protocol), relaxed memory consistency models, prefetching, and multiple hardware contexts. The first two techniques exhibit consistent performance improvements across all the studied benchmarks. Therefore, we have considered both these techniques in our baseline architecture *BASIC*. Although this combination can eliminate all the write penalty, the read penalty is a severe problem.

One of the protocol extensions we have considered is a hardware-based sequential prefetching scheme. Hagersten's ROT prefetching [14] is another prefetching scheme which takes advantage of the regularity of data accesses in scientific computations by dynamically detecting access

strides. Unfortunately, this solution requires complex hardware and does not work well for applications with irregular strides [5]. Another proposal by Lee et al. [15] relies on data address lookahead in the processor, but is limited by the number of instructions in basic blocks. Mowry and Gupta studied software-based prefetching [16]. Special prefetch instructions are explicitly inserted in the code (by the programmer or by the compiler).

Finally, we did not consider processor multithreading because Gupta et al. [13], using a similar benchmark suite, clearly showed that its interaction with prefetching is complex and oftentimes degrades performance. Instead we have considered previously proposed cache protocol optimizations and studied their interactions.

In our study, we have assumed single issue, statically scheduled processors. Today, processors exploit instruction-level parallelism to a continuously increasing extent, including multiple-issue, dynamic scheduling and nonblocking reads. Gharachorloo et al. [11] explored the consequences of hiding read latency through relaxed memory models. They found that it would require very large instruction windows and very accurate branch prediction to be effective. In [17], Pai et al. explore the impact of instruction-level parallelism (ILP) on multiprocessor performance. In their evaluation, they include three of the applications we have used in this paper; LU, MP3D, and Water. Their results show that, while the read stall time in wall clock time is often shorter for a system with ILP processors as compared to the same system with scalar, statically scheduled processors, the relative speedup of the busy time, i.e., the time spent on actual instruction execution, was much larger. As a result, a system with ILP processors spends a larger fraction of its execution time stalled because of latencies in the memory system. This makes us believe that the performance improving techniques explored in this paper are at least as important for ILP processors.

7 Conclusions

In this paper, we have evaluated the combined performance gains and hardware costs of three simple extensions to a directory-based write-invalidate protocol: adaptive sequential prefetching (*P*), a migratory sharing optimization (*M*), and a competitive-update mechanism (*CW*). These extended protocols and all their combinations were shown to add only marginally to the complexity of the second-level caches and of the system-level cache coherence protocol. Moreover, since they are hardware-based, they do not impose any requirement on software.

Out of the four possible combinations, we have found *P* + *CW* and *P* + *M* to be particularly effective. They often provide additive performance gains because they attack different components of the processor penalties; e.g., while *P* cuts the read penalty, *M* cuts the write penalty, resulting in a combined gain of nearly a factor of two for some applications under sequential consistency. Moreover, we did not see any detrimental effects of the interactions between these individual techniques.

P + *CW* provides a performance improvement of nearly a factor of two under release consistency for Cholesky, assuming enough network bandwidth. Network contention has virtually no effect except for conservative mesh network implementations. Although the prefetching mechanism and the write caches in *P* + *CW* cost additional hardware, we have shown that the buffer space needed in the lockup-free cache is almost the same as for *BASIC*. Thus, given a reasonable network bandwidth, we believe that *P* + *CW* is an important technique.

Unlike *P* + *CW*, *P* + *M* is applicable to sequential consistency as well as to release consistency. Under sequential consistency, *P* + *M* was shown to improve performance significantly for all applications. For three out of five applications, it even outperformed our baseline architecture under release consistency. Under release consistency, although *P* + *M* does not improve performance to the same extent as does *P* + *CW*, it was shown to be less sensitive to network contention. The optimization for migratory sharing (*M*) reduces the write traffic generated, which can be exploited by prefetching. Thus, *P* + *M* opens the possibility of utilizing prefetching with high-contention network. To our knowledge, *P* + *M* is the first example of *hardware-based read-exclusive prefetching*.

This paper shows that a basic write-invalidate protocol augmented by appropriate extensions can eliminate most memory access penalties without any support from the programmer or the compiler. An open and interesting direction for future research is to investigate which penalties can be attacked by a combination of compiler algorithms and simple hardware mechanisms to meet the demands of next generation high-performance processors.

References

[1] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, "The CacheMire Test Bench—A Flexible and Effective Approach for Simulation of Multiprocessors," *Proc. 26th Ann. Simulation Symp.*, pp. 41-49, 1993.

[2] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1,112-1,118, Dec. 1978.

[3] A.L. Cox and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp.98-108, 1993.

[4] F. Dahlgren, M. Dubois, and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733-746, July 1995.

[5] F. Dahlgren and P. Stenström, "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385- 398, Apr. 1996.

[6] F. Dahlgren, M. Dubois, and P. Stenström, "Combined Performance Gains of Simple Cache Protocol Extensions," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, pp. 187-197, 1994.

[7] F. Dahlgren and P. Stenström, "Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 26, no. 2, pp. 193-210, Apr. 1995.

[8] M. Dubois and C. Scheurich, "Memory Access Dependencies in Shared Memory Multiprocessors," *IEEE Trans. Software Eng.*, vol. 16, no. 6, pp. 660-674, June 1990.

[9] S.J. Eggers and R.H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, pp. 373-382, 1988.

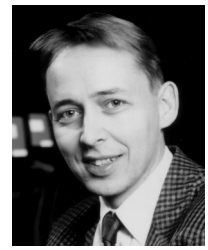
[10] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 245-257, 1991.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, "Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 22-33, 1992.
[12] H. Grahn, P. Stenström, and M. Dubois, "Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models," *Future Generation Computer Systems*, vol. 11, no. 3, pp. 247-271, June 1995.
[13] A. Gupta et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 254-263, 1991.
[14] E. Hagersten, "Towards Scalable Cache Only Memory Architectures," PhD thesis, Swedish Inst. of Computer Science, Oct. 1992 (SICS Dissertation Series 08).
[15] R. Lee, P.-C. Yew, and D. Lawrie, "Data Prefetching in Shared-Memory Multiprocessors," *Proc. 1987 Int'l Conf. Parallel Processing*, vol. I, pp. 28-31, 1987.
[16] T. Mowry and A. Gupta, "Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 2, no. 4, pp. 87-106, June 1991.
[17] V.S. Pai, P. Ranganathan, and S.V. Adve, "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology," *Proc. Third Int'l Symp. High-Performance Computer Architecture*, pp. 72-83, 1997.
[18] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5-44, Mar. 1992.
[19] P. Stenström, T. Joe, and A. Gupta, "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 80-91, 1992.
[20] P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp. 109-118, 1993.

Fredrik Dahlgren received his MS degree in computer science and engineering in 1990, and his PhD degree in computer engineering in 1994, both from Lund University. He is an assistant professor in the Department of Computer Engineering, Chalmers University of Technology. His research is focused on computer architecture, with special emphasis on the interaction between application and architecture in shared-memory multiprocessors, memory systems, and performance evaluation techniques. As a visiting scientist at the Massachusetts Institute of Technology, he participated in the multiprocessor research during 1995. He is a member of the IEEE Computer Society. More information about Dahlgren's current activities can be found at http:// www.ce.chalmers.se/~dahlgren/.

Michel Dubois holds a PhD from Purdue University, an MS from the University of Minnesota, and an engineering degree from the Faculte Polytechnique de Mons in Belgium, all in Electrical Engineering. He is a professor in the Department of Electrical Engineering of the University of Southern California. Before joining USC in 1984, he was a research engineer at the Central Research Laboratory of Thomson-CSF in Orsay, France. His main interests are computer architecture and parallel processing, with a focus on multiprocessor architecture, performance, and algorithms. He currently leads the RPM Project. RPM is a flexible hardware platform built with FPGAs and used to prototype multiprocessor systems with widely different architectures. More information can be found on the world wide web at http://www.usc.edu/dept/ceng/dubois/RPM.html and at http://www.usc.edu/dept/ceng/dubois/dubois.html.

Dr. Dubois is a member of the ACM and a senior member of the Computer Society of the IEEE.



Per Stenström has been a professor of computer engineering at Chalmers University of Technology since 1995. He was previously on the faculty of Lund University, where he also received his MS degree in electrical engineering and a Ph.D. degree in computer engineering in 1981 and 1990, respectively. Dr. Stenström's research interests are in computer architecture in general, with an emphasis on multiprocessor design and performance analysis, as well as compiler optimization techniques. He has published more than 50 papers in these areas and has authored two textbooks on computer architecture and organization. As a visiting scientist, he participated in major multiprocessor architecture research projects at Carnegie Mellon University, Stanford University, and the University of Southern California. He is on the editorial board of the *Journal of Parallel and Distributed Computing* (*JPDC*) and has served on 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 and the ACM. For more information about Stenström's current activities, please refer to http://www.ce.chalmers.se/~pers.