Combined Performance Gains of Simple Cache Protocol Extensions

Fredrik Dahlgren, Michel Dubois*, and Per Stenström

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

*Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA90089-2562, U.S.A.

Abstract

We consider three simple 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 they target different components of the read and write penalties, they can be combined effectively.

Detailed architectural simulations using five benchmarks show substantial combined performance gains obtained at a modest additional hardware cost. Prefetching in combination with competitive-update is the best combination under release consistency in systems with sufficient network bandwidth. By contrast, prefetching plus the migratory sharing optimization is advantageous under sequential consistency and/or in systems with limited network bandwidth.

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 recent studies have focused on techniques to tolerate or reduce the latency. In [8] Gupta *et al.* compared three approaches to tolerating latency, which are 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. The success of software-based prefetching is predicated on effective algorithms to predict cache misses at compile time. Multithreading is only applicable to specially designed processors with more than one context and it requires more application concurrency.

By contrast, in this paper, we evaluate simple extensions to the cache protocol itself. These extensions are completely hardware-based, are transparent to the software, and are compatible with traditional, single-context processors. Moreover, they add only marginally to the overall system complexity while still providing a significant performance advantage when applied separately. We focus on three extensions, *adaptive sequential prefetching* [3], *a migratory sharing optimization* [2,12] and *a competitive-update mechanism* [10,4].

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 [3]. The migratory sharing optimization is aimed at reducing the write penalty for migratory blocks, a major component of the overall memory access penalty under sequential consistency. Although this optimization requires only minor modifications to the cache coherence protocol, two independent studies [2,12] have shown that it is very successful in many cases.

The above two protocol extensions are fairly ineffective when it comes to coherence misses. On the other hand, it is well-known that write-update protocols have no coherence miss penalty. The major drawback of these protocols is their large write memory traffic. *Competitive-update* protocols trade off traffic and penalty by mixing updates and invalidations. The idea is simple. Instead of invalidating a block copy at the first write by another pro-

This work was partly supported by the Swedish National Board for Technical Development (NUTEK) under contract number 9001797 and by the National Science Foundation under Grant No. CCR-9115725.

1063-6897/94 $03.00 © 1994 IEEE

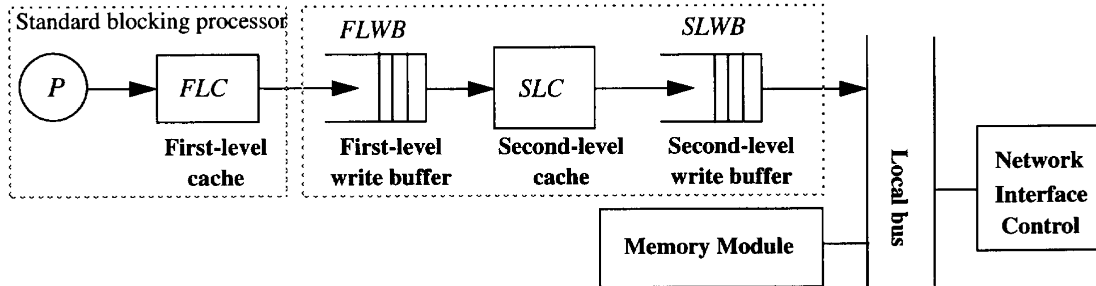

Figure 1: Organization of the memory subsystem in each processor node.

cessor, 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 [10]. To further reduce the write traffic, a small *write cache* [4] 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 effects 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. We find that the combined techniques provide *additive gains* in many cases. For example, in many cases, the performance improvement of adaptive prefetching combined with the competitive-update mechanism is the sum of the individual improvements provided by each technique; in some cases, this combination yields a speedup close to two under release consistency. These gains are obtained by simple modifications to the *lockup-free* caches [6] and to the coherence protocol.

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 [11] introduced in Section 4, we evaluate the combined performance gains of the protocol extensions in Section 5. Finally, we compare the evaluations in this paper with work done by others in Section 6, before concluding the paper in Section 7.

2 Framework and Baseline Architecture

None of the three protocol extensions we consider sets any specific requirement on the processor architecture. Standard, off-the-shelf processors with blocking loads will do. Figure 1 shows the overall organization of each processing node. The environment of each processor is comprised of a first-level cache (*FLC*), a second-level cache (*SLC*), a first-level write buffer (*FLWB*), and a second-level write buffer (*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. Write and read miss requests issued by the *FLC* are buffered in FIFO order in the *FLWB*. The second-level cache (*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*.

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* [6] 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 [6] such as *release consistency* [7]. 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. 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* node (the node where 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.

A memory block can be in two stable states: CLEAN (the memory copy is valid) and MODIFIED (exactly one cache keeps the exclusive copy of the memory block). In addition, three transient states are also encoded. For example, while the home node is waiting for the completion of a coherence action such as the invalidation of cached copies, the memory block is in a transient state. Three bits per memory block are needed to encode these five states.

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 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 [7]. This overlap is achieved through the lockup-free mechanism implemented in the *SLC* controller working in conjunction with the *SLWB*.

3 Simple Cache 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 to combine the protocol extensions to further cut the penalties, in Section 3.4.

3.1 Adaptive Sequential Prefetching

Non-binding prefetching [8] 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 [9], whereas hardware prefetching schemes dynamically detect patterns in past and present data accesses to predict future accesses. Non-binding prefetching is applicable to systems under any memory consistency model, including sequential consistency (SC) and release consistency (RC).

A non-binding hardware-based prefetching scheme called *adaptive sequential prefetching* was proposed and evaluated experimentally in [3]. 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. 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, and are pipelined in the memory system with the original miss request.

This prefetching scheme exploits spatial locality across cache blocks. Although one might think that a block K times larger could have the same effect, it was shown in [3] 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 [3]. If K is allowed to vary during the execution of a program, sequential prefetching was shown to be significantly more effective.

Conceptually, the adaptive prefetching scheme controls the degree of prefetching 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. More details of this scheme are given in [3].

The adaptive prefetching scheme improves performance by cutting the number of cache misses. From the simulation of six benchmark programs from the SPLASH suite [11] (five of them are used in this paper), 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., non-iterative) solution methods in linear algebra, exemplified by LU and Cholesky 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 applications was also reported in [3], 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 the spatial locality in the true-sharing miss component, which is exploited without increasing the false-sharing miss component.

Table 1: Hardware needed to support *BASIC* and extra hardware needed by each extension.

	BASIC	P (overhead)	M (overhead)	CW (overhead)
State bits per *SLC* line	2 bits (3 states)	2 bits	1 state	1-bit counter
Additional mechanisms per cache	None	3 counters (4 bits)	None	Write cache with four blocks
***SLWB* features**	SC: a single entry RC: several entries	Prefetch requests are buffered in the *SLWB*	None	Each entry holds a block
State bits per memory line	3 state bits plus N presence bits	No extra state	1 state bit plus a pointer ($log_2 N$ bits)	No extra state

The hardware needed to extend *BASIC* with the adaptive prefetching scheme consists of 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

Whereas 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* 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 [2] and Stenström *et al.* [12] 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 [2,12]). [12] evaluates the performance advantage of the migratory optimization. For three applications (MP3D, Cholesky, and Water), the number of ownership requests are cut by between 69% and 96% and the execution time is reduced by as much as 35% (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, and the ensuing reduction of memory and network 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 to keep track of migratory blocks, a pointer of size $log_2 N$ bits (for N caches) per memory line, and an extra state per cache line [12]. This extra cache state is needed to disable the migratory optimization when the access pattern to a block changes from migratory to another form of sharing such as read-only sharing. These overheads appear in column M in Table 1.

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, at the cost of an 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 memory conflicts caused by the intense write traffic may offset the gains.

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 in the context of architectures similar to *BASIC* [10]. Their6 hardware cost is limited to a counter associated with each cache line. When a block is loaded into the cache, or when the cache block is accessed, the counter is preset to a value called the *competitive threshold*. The counter is decremented every time the cache 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 better performance of a competitive-update protocol over a pure write-invalidate protocol [4,10]: (i) the coherence miss rate is reduced and (ii) the latencies of the remaining coherence misses are shorter because the likelihood of finding a clean copy at memory is higher. In [10], 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 [10] as a good overall compromise.

This simple competitive-update mechanism is even more effective in conjunction with *write caches* [4]. 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 in which the propagation of writes can be delayed until a synchronization point. For example, under release consistency, the propagation of updates to a block in the write cache can wait until the write-cache block is replaced or until the release of a lock.

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. If a read misses in the *SLC* but hits in the write cache the data is returned to the processor from the write cache. However, if a read misses in both the *SLC* and 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. The dirty bits are also used to selectively send the modified words to the home node using a single request to further reduce memory traffic. Write-cache blocks transit through the *SLWB* on their way to memory.

Detailed performance evaluations reported in [4] 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 in the write cache, 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). Each entry of the *SLWB* is a block rather than a word but the number of entries in the *SLWB* is less than in *BASIC*. These overheads are summarized in column *CW* in Table 1.

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 misses to blocks deemed migratory retrieve exclusive copies, prefetching in *P+M* is equivalent to a hardware-based *read-exclusive prefetching scheme* [9]. 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. First, since *CW* is only applicable to implementations under relaxed memory consistency models, the gain reaped by the addition of *M* to *CW* comes from the reduced traffic due to migratory sharing, which

may be critical if an application requires more bandwidth than the network can deliver.

The competitive-update mechanism affects migration detection because the home node only sees updates and not local reads and it cannot detect that two consecutive non-overlapping read/write sequences by distinct processors are migratory. To make sure that read/write sequences are non-overlapping, 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 Benchmarks

We have developed simulation models of *BASIC* and of 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 processors. It consists of two parts: (i) a functional simulator and (ii) 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*.

In our simulations, the number of processors is 16, the *FLC* size is 4 Kbytes and the *SLC* is infinite with a block size of 32 bytes. The processors and *FLC* are clocked at 100 MHz (1 pclock = 10 ns). An *FLC* block fill on a miss takes 3 cycles. The *SLC* is built from static RAMs with a cycle time of 30 ns. The memory in each processor node is fully interleaved with an access time of 90 ns, and the 256-bit wide local split-transaction bus is clocked at 33 MHz. These design parameters lead to *FLC*, *SLC*, and local memory access times of 1, 6, and 30 pclocks, respectively.

By default, we assume a contention-free uniform access time network with a node-to-node latency of 54 pclocks, although contention is accurately modelled in each node. To specifically study the impact of network contention, however, 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, the memory pages of size 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. Three of them are taken from the SPLASH suite (MP3D, Water, and Cholesky) [11]. 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 with `gcc` (version 2.1). For all measurements, we gather statistics during the parallel sections only in accordance with the recommendations in the SPLASH report [11].

MP3D was run with 10K particles for 10 time steps. Cholesky used the `bcsstk14` matrix. Water was run with 288 molecules for 4 time steps. A 200x200 matrix was the input to LU. Finally, Ocean used a 128x128 grid with the tolerance factor set to 10^{-7}.

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 sufficient network bandwidth and local buffering under release consistency. In Sections 5.2 through 5.4, we impose various architectural constraints: sequential consistency in Section 5.2, network bandwidth limitations in Section 5.3, and buffer and cache size limitations in Section 5.4.

5.1 Individual and Combined Performance Gains

The baseline architecture for our performance comparisons is *BASIC* under release consistency (RCpc as defined in [7]). *BASIC* exploits the write latency tolerance afforded by release consistency with a 16-entry deep *SLWB* and an 8-entry deep *FLWB*. Henceforth, unless otherwise stated, *BASIC* will refer to this implementation.

Latency tolerance techniques in general consume network bandwidth. In order to estimate the maximum possible gains given enough network bandwidth, we have run simulations for infinite network bandwidth. Contention in each node is modelled carefully. Later in Section 5.3, we will consider the impact of network contention in detail.

Figure 2 shows the execution times for each benchmark program relative to *BASIC*. The execution time for each protocol extension is decomposed into the fraction of busy time, read stall time, and acquire stall time (the time spent waiting for an acquire to complete). Being completely hid-

den, the write latency 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. The read stall time in *P* is reduced because of less cache misses for all applications except Ocean. To see this, we show in Table 2 the cold and coherence miss rates for *P* and for *BASIC*. For example, *P* cuts the cold miss rate from 0.86% to 0.22% in LU, and from 0.90% to 0.19% in Cholesky. *P* also removes some coherence misses in MP3D, Cholesky, and Water.

Table 2: Cold and coherence miss rate components (in percent) for *BASIC, P, CW,* and *P+CW*.

	BASIC		P		CW		P + CW	
Appl.	Cold	Coh.	Cold	Coh.	Cold	Coh.	Cold	Coh.
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

The protocol with the competitive-update mechanism (*CW*) uses a competitive threshold of one and a write cache of four blocks. The read stall time is reduced substantially in *CW* for all applications but LU because of the reduction of the coherence miss penalty. Table 2 confirms this expectation. Whereas the coherence miss rates are cut by *CW* for all applications, the cold miss rates are virtually unaffected. In the case of MP3D, *CW* only manages to reduce the coherence miss rate from 9% to 8% (see Table 2). Nonetheless, the read penalty reduction is significant and is essentially due to the shorter latency of the remaining coherence misses, which are serviced mostly by the home node because the memory copy is more often clean than in *BASIC*. We measured the average time to handle a read miss for MP3D and found that it is 41% 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

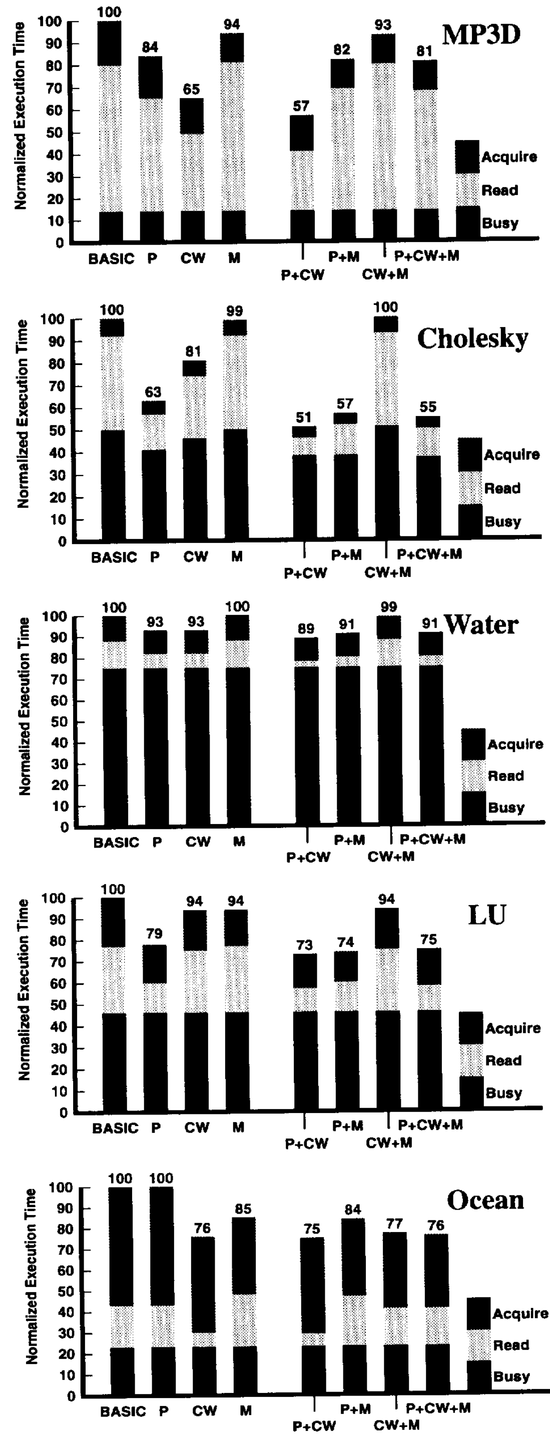

Figure 2: Execution times relative to *BASIC* under release consistency.

helps cut the acquire stall time, as is clearly visible in the case of MP3D, LU, and Ocean (see Figure 2).

Combining *P* and *CW* (*P+CW*) leads to further reductions of the read stall times beyond *P* or *CW* alone. For the sake of illustration, consider the case of LU in Figure 2. 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 2, 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 2.) 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% and 81% of the read stall time for all applications.

P+M performs only slightly better than *P* alone because, under release consistency, the contribution of *M* is limited to cutting the acquire stall time. 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, the effect of *CW* is to keep 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 with enough network bandwidth. The combination of all three techniques (*P+CW+M*) has practically 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, *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 very useful because there is no write penalty under release consistency.

5.2 Sequential Consistency

An advantage of release consistency is the complete elimination of the write penalty. However, sequential consistency offers a more intuitive programming model and may be preferred in some environments. Techniques to reduce the read stall time *as well as* the write stall time under sequential consistency are therefore important. 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*. Under *BASIC* and *M*, a single entry is needed in the *SLWB* whereas, in *P*, the *SLWB* must keep track of pending prefetch requests. Since the designs of the *SLC* and the *SLWB* under sequential consistency are different from the release consistency implementations we denote *BASIC* and *M* under sequential consistency as *B-SC* and *M-SC*, respectively. In the case of *P* the designs are the same and the distinction is not made.

Figure 3 shows the execution times for *P*, *M-SC*, and *P+M* under sequential consistency relative to *B-SC* 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 degradation of the write stall time is a side effect of prefetching, which increases the number of cached copies and consequently causes the propagation of more invalidations. 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 maximum execution time reduction with respect to *B-SC* occurs for Cholesky and is limited to 26%

Since *M-SC* is aimed at the write and acquire stall times for migratory blocks, it is very effective in the cases of 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. Some ownership requests are removed and thus the write and acquire stall times are cut, but the read miss rate and the read stall time tend to increase. Overall, the execution time reduction in *M-SC* with respect to *B-SC* is limited to 39% and is observed in the case of MP3D.

We have seen that *P* improves the read stall time and that *M* reduces the write and the acquire stall times. Therefore, the combination of the two techniques might reduce all three components. Figure 3 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 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*. The combined gains lead to an execution time reduction of 46% for MP3D and 55% for Cholesky. *P+M* is, to 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 modifica-

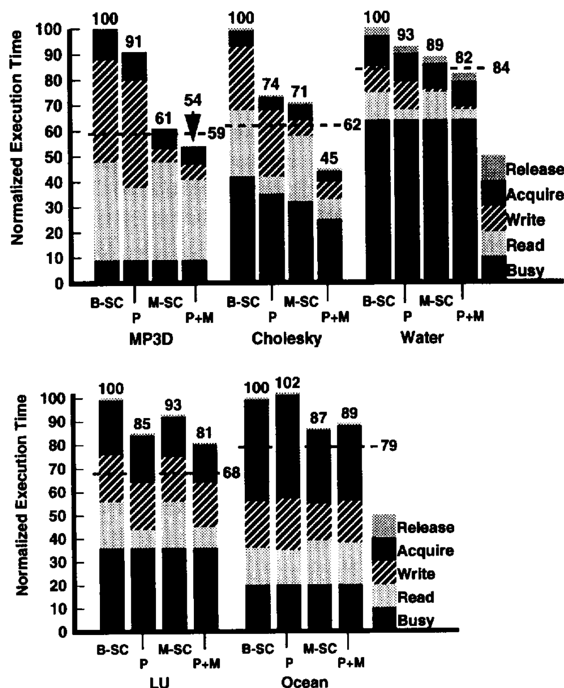

Figure 3: Execution times for *B-SC, P, M-SC,* and *P+M* under sequential consistency. The dashed line corresponds to *BASIC* under release consistency.

tion 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, Figure 3 shows that these effects are negligible.

Finally, to compare the execution times under sequential consistency and under release consistency, we show the execution time for *BASIC* (using a *SLWB* with 16 entries) in the diagrams of Figure 3 (dashed lines). We observe that 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 Network Contention

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 mesh networks having link widths 64, 32, and 16 bits. The meshes are wormhole-routed with two phases (routing + transfer), and are clocked at the same frequency as the processors (100 MHz). These network implementations are not overly aggressive and thus pin-point the contention effects of the cache protocol extensions.

The ratios between the execution times of *P+CW* and of *BASIC* and between the execution times of *P+M* and *BASIC* for the cases of the three meshes under release consistency are given in Table 3. For example, consider MP3D. With a 64-bit mesh, the execution time under *P+CW* is only 69% of the execution time under *BASIC*. However, with 16-bit links, the execution time under *P+CW* is 9% longer than under *BASIC*. We conclude that the relative advantage of *P+CW* over *BASIC* for MP3D is higher in systems with wider links.

Table 3 : Impact of contention on the execution-time ratio (ETR) for *P+CW* and *P+M*.

	Links	MP3D	Cholesky	Water	LU	Ocean
P+CW	64	0.69	0.69	0.94	0.88	0.81
	32	0.87	0.75	0.94	0.88	0.81
	16	1.09	1.00	0.94	0.93	0.82
P+M	64	0.87	0.71	0.95	0.87	0.90
	32	0.90	0.72	0.95	0.88	0.90
	16	0.92	0.80	0.94	0.91	0.92

In order to clarify the effect of network bandwidth on the execution time, we show in Figure 4 the total amount of network traffic generated in different systems under release consistency normalized to the traffic in *BASIC*. For MP3D the traffic is 36% 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 (Table 3): in systems with 32-bit and 64-bit links, the execution time is 75% and 69% of that under *BASIC* but these gains vanish in systems with 16-bit links. Figure 4 reveals that the total amount of network traffic generated by Cholesky in *P+CW* is about 70% 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 the network with 16-bit links saturates.

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 memory traffic and is already affected by higher traffic in systems with 32-bit links; on the other

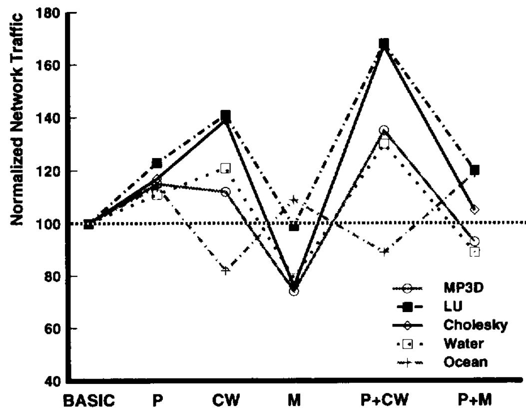

Figure 4: Total network traffic generated by the protocols and normalized to the traffic in *BASIC*.

hand, the performance of Water is fairly 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*, suggesting that *P+CW* is a very good combination at reasonable network bandwidths.

In contrast with *P+CW*, the difference in network traffic between *P+M* and *BASIC* is small for all applications, as can be observed from Figure 4 and the relative improvements of *P+M* over *BASIC* is not very sensitive to network contention. This is confirmed in Table 3, in which the sensitivity of the relative execution times to the link width is very small. Compared to *P+CW*, the gains of *P+M* are only slightly affected by network contention. As a result, *P+M* is a promising approach for hardware prefetching under limited network bandwidth.

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 becomes available to the prefetching mechanism.

5.4 Sensitivity to Architectural Parameters

We have also analyzed the sensitivity of the performance gains to some key architectural parameters. Due to space limitations, we only present the main results. A complete presentation is found in [5].

In order to hide the write latency, *BASIC* uses a *SLWB* that keeps track of write requests to transient blocks in the *SLC*. Similarly, *P* and *CW* need multiple *SLWB* entries to keep track of outstanding prefetch and update requests, respectively. By contrast, *M* is expected to need fewer *SLWB* entries than *BASIC* because the migration optimization cuts the number of global invalidation requests. To study how sensitive each extension is to the size of the *SLWB*, we have rerun the experiments in Section 5.1 with smaller *FLWB* and *SLWB* of 4 entries each. We found that only *BASIC* and *P* suffered to some extent due to a limited buffer space. However, the penalty is not attributable to prefetching but to pending write requests. By contrast, the reduced buffer sizes did not impact *CW*, *M* or any combination including these techniques. For example, *P+CW* and *P+M* need less complex *SLWB*s than *BASIC*.

We have also analyzed the performance gains assuming a limited (16 Kbytes) direct-mapped *SLC*. The overall conclusion of these experiments is that the combinations yielding substantial gains with infinite caches did so too with limited caches. In fact, *P* is very effective in many cases for eliminating replacement misses and its gains are even better in systems with limited cache sizes.

6 Related Work

Gupta *et al.* compared four latency tolerance and reduction techniques [8] consisting of coherent caches (with a write-invalidate protocol), relaxed memory consistency models, software-based prefetching, and multiple hardware contexts. The first two techniques exhibit consistent performance improvements across all the studied benchmarks. Therefore, we have considered both techniques in our baseline architecture *BASIC*. Although this combination can eliminate all the write penalty, the read penalty is a severe problem. The read penalty could also be tolerated or reduced using relaxed memory consistency models or multithreading. However, this would require that the processors support non-blocking loads and/or multiple contexts. By contrast, our protocol extensions are compatible with processors blocking on loads and supporting a single context.

The hardware-based sequential prefetching scheme we have simulated is radically different from Mowry and Gupta's software-based prefetching [9], in which special prefetch instructions are explicitly inserted in the code (by the programmer or by the compiler). Although we considered only one prefetching technique, we have no reason to believe that other hardware-based or software-based prefetching schemes would interact with *M* and *CW* in a qualitatively different fashion.

7 Conclusions

In this paper we have evaluated the combined performance gains and hardware complexity of three simple extensions to a directory-based write-invalidate protocol, which are 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.

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; for example, whereas *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. In fact, for three out of five applications *P+M* even outperformed our baseline architecture under release consistency. Moreover, we did not see any detrimental effects due to interactions between these individual techniques. Similarly, while *P* mainly cuts the read penalty due to cold misses, *CW* can reduce the coherence-miss penalty and *P+CW* shows a speedup of nearly a factor of two under release consistency.

We also analyzed the network bandwidth required by each individual extension as well as their combinations. Although *P+M* does not improve performance to the same extent as *P+CW* does, it was shown to be less sensitive to network contention. The optimization for migratory sharing (*M*) reduces the write traffic and the bandwidth can be exploited by prefetching. Thus, *P+M* opens the possibility of utilizing prefetching for networks with a limited bandwidth. This technique is also the first example of *a hardware-based read-exclusive prefetching scheme.*

This paper shows that a basic directory-based write-invalidate protocol augmented by appropriate extensions can eliminate a substantial part of the memory access penalty without significantly increasing the complexity of neither the hardware design nor the software system.

References

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

[2] Cox, A.L. and Fowler, R.J. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. of the 20th Annual Int. Symp. on Computer Architecture*, pp.98-108, May 1993.

[3] Dahlgren, F., Dubois, M., and Stenström, P. Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors. In *Proc. of 1993 Int. Conf. on Parallel Processing*, Vol. I, pp. 56-63, 1993.

[4] Dahlgren, F. and Stenström, P. *Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors*. Tech. Rep., Dept. of Comp. Eng., Lund University, April 1993. Presented at the *Third Workshop on Scalable Shared-Memory Multiprocessors*, May 1993.

[5] Dahlgren, F., Dubois, M., and Stenström, P. *Performance Gains and Cost Trade-off for Cache Protocol Extensions*. Tech. Rep. Dept. of Comp. Eng., Lund University, Feb. 1994.

[6] Dubois, M. and Scheurich, C. Memory Access Dependencies in Shared Memory Multiprocessors. In *IEEE Trans. on Software Engineering*, 16(6), pp. 660-674, June 1990.

[7] Gharachorloo, K., Gupta, A., Hennessy, J. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. of ASPLOS IV*, April 1991.

[8] Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T., and Weber, W.-D. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th Ann. Int. Symp. on Computer Architecture*, pp.254-263, May 1991.

[9] Mowry, T. and Gupta, A. Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors. In *Journal of Parallel and Distributed Computing*, 2(4), June 1991.

[10] Nilsson, H., Stenström, P., and Dubois, M. *Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models.* Tech. Rep. Dept. of Comp. Eng., Lund University, July 1993.

[11] Singh, J.P., Weber, W.-D., and Gupta, A. SPLASH: Stanford Parallel Applications for Shared-Memory, In *Computer Architecture News*, 20(1):5-44, March 1992.

[12] Stenström, P., Brorsson, M., and Sandberg, L. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of the 20th Ann. Int. Symp. on Computer Architecture*, pp.109-118, May 1993.