In *Future Generation Computer Systems*, Vol. 11, No. 3, pages 247-271, June 1995.

Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models[1]

Håkan Grahn, Per Stenström, and Michel Dubois*

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden
Email: nesse@dit.lth.se
Fax: +46-46-104-714

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

Abstract

Invalidation-based cache coherence protocols have been extensively studied in the context of large-scale shared-memory multiprocessors. Under a relaxed memory consistency model, most of the write latency can be hidden whereas cache misses still incur a severe performance problem. By contrast, update-based protocols have a potential to reduce both write and read penalties under relaxed memory consistency models because coherence misses can be completely eliminated. The purpose of this paper is to compare update- and invalidation-based protocols for their ability to reduce or hide memory access latencies and for their ease of implementation under relaxed memory consistency models.

Based on a detailed simulation study, we find that write-update protocols augmented with simple competitive mechanisms — we call such protocols *competitive-update* protocols — can hide all the write latency and cut the read penalty by as much as 46% at the cost of some increase in the memory traffic. However, as compared to write-invalidate, update-based protocols require more aggressive memory consistency models and more local buffering in the second-level cache to be effective. In addition, their increased number of global writes may cause increased synchronization overhead in applications with high contention for critical sections.

[1]This research was partially supported by the Swedish National Board for Industrial and Technical Development (Nutek) under contract 9001797 and by the U.S. National Science Foundation under Grant No. CCR.9115725.

1 Introduction

Shared-memory multiprocessors do not easily scale to large numbers of processors because of the latency of accesses to shared data. Using private caches with a directory-based write-invalidate cache coherence protocol is a common approach to reduce these latencies [30]. However, as faster processors are designed, cache coherence and miss handling can significantly reduce processor efficiency. Several latency-tolerating and hiding techniques have been proposed and evaluated [19] including prefetching [8, 10, 26], multiple hardware contexts [2], and memory access buffering under relaxed memory consistency models [1, 13, 16].

Previous studies have shown that under relaxed memory consistency models the write latency can be easily hidden by overlapping write requests with each other and with local computation. Gharachorloo *et al.* [17] studied the effectiveness of hardware mechanisms under memory consistency model relaxation to hide write latency in the context of processors with blocking loads. Hiding read latency has also been studied in two subsequent papers by Zucker and Baer [36] and by Gharachorloo *et al.* [18] by considering processors that do not block on load accesses. Unfortunately, the tolerance to read miss latency by relaxing the memory consistency model can be severely restricted by the limited ability to schedule loads sufficiently far ahead of the miss or by the hardware complexity needed in the processor to dynamically schedule the loads.

These observations have motivated us to evaluate update-based cache protocols, which maintain consistency by propagating the data values on each shared write, in the context of standard blocking load processors. Update-based protocols trade a reduction in the miss rate for an increase in the write traffic. Unfortunately, several problems may eliminate this performance advantage. First, update-based protocols generate more write traffic and block the processors on a write more often than invalidation-based protocols; therefore more aggressive hardware mechanisms and memory consistency models may be needed to hide the write latency. Second, even if the write latency can be hidden, the larger write traffic can cause network contention which, as a secondary effect, may increase the latency of misses; to offset this effect, higher bandwidth networks may be required.

In this study, we quantify these performance effects to compare update-based to invalidation-based protocols. As a basis for the comparison, we consider a two-level cache hierarchy in each processor node consisting of a simple and fast write-through, direct-mapped first-level cache interfaced to a second-level write-back cache with various degrees of write buffering. We especially focus on the implementation issues related to a lockup-free [23, 31] second-level cache. A previous study by Gharachorloo *et al.* has partly addressed this issue in the context of write-invalidate protocols [17] but not for update-based protocols.

A detailed simulation study of four applications from the SPLASH suite [29] reveals that pure write-update protocols have an unacceptable level of traffic in some cases. However, *competitive-update* protocols, which are update-based protocols augmented with simple competitive mecha-

nisms to invalidate a block, have a potential to reduce the read penalty provided that the application's bandwidth requirement is moderate as compared to the available network bandwidth. We show that competitive-update protocols can reduce the read penalty by as much as 46% as compared to write-invalidate protocols. However, the management of the cache for competitive-update protocols requires more complex hardware and is only effective under relaxed memory consistency models. We identify the hardware mechanisms needed to fully exploit the read-latency reducing capability of update-based protocols.

As a background, we begin in the next section by comparing the performance potentials and limitations of write-invalidate and update-based protocols under relaxed memory consistency models. Then, in Section 3, we describe the lockup-free mechanisms needed to hide the write latency by reviewing the architecture of the simulated multiprocessor system used in our simulations. In Section 4 we present the simulation methodology, the detailed architectural assumptions, and the benchmark programs. The experimental results are presented in Section 5 and our findings are contrasted with the work of others in Section 6. Finally, we conclude in Section 7.

2 Background

In shared-memory multiprocessors, the memory access penalty, i.e., the accumulated time the processor has to stall for completing memory accesses during the program execution, consists of two components — the *write* and the *read penalties*. In a write-invalidate protocol, the write penalty is due to the invalidation of remote copies upon a write request whereas the read penalty comes from the handling of cache load misses. Write latencies can be very high because copies in remote caches, far away from the processor, must sometimes be invalidated. The effectiveness of mechanisms to hide these latencies depends on the *memory consistency model*.

2.1 Memory Consistency Models

The memory consistency model refers to the logical model offered by the memory system to the programmer or to the compiler. This model in turn constrains the possible ordering and interleaving of memory accesses in the multiprocessor.

Sequential Consistency [24] is the most restrictive model as far as the ordering of memory accesses is concerned. The major drawback of sequential consistency is the severe limitation it imposes on the overlap of writes with subsequent reads, writes, or local computation in the processor. In essence, no read or write request (to shared data) can be handled before a previous write request has been completed.

To remedy this problem the constraints on the ordering of shared memory accesses [1, 13, 16] must be relaxed by assuming that ordering is only enforced on special synchronization operations rather than on all memory accesses. All synchronizations among parallel threads are done through explicit, hardware-recognizable synchronization operations (i.e., these operations must be distinguishable from regular load/store instructions). The processor must perform all its preceding

loads and stores globally before it can issue a synchronization operation; moreover, a processor may issue no memory loads or stores following a synchronization point in program order until the synchronization operation is successfully completed. In systems where these two conditions apply we say that loads and stores are *Weakly Ordered* and that the memory system is weakly ordered (WO) [12].

Special types of synchronization operations allow additional relaxation of the above conditions. For synchronization based on critical sections, a refinement called *Release Consistency* [16] distinguishes between acquire (acquiring a lock) and release (releasing a lock). Release Consistency requires that all global accesses preceding a release are globally performed before the release, and that no global access following an acquire is issued before the acquire has completed. In its strictest form, Release Consistency (RCsc) requires that all processors must execute their acquires and releases in their program order. In essence, releases can be buffered with the stores in the same write buffer provided all writes preceding a release in the FIFO order of the buffer are performed before the release is issued from that buffer and acquires are not allowed to bypass releases. In a more relaxed form of Release Consistency (RCpc), acquires are allowed to bypass previous releases, but consecutive releases must still be performed in program order. Henceforth we will only assume RCpc and will refer to it as RC for simplicity.

Under Release Consistency, previous work has shown that there is a potential to hide all the write latency by local computation given enough hardware support [17]. However, the read penalty cannot be reduced significantly by using relaxed memory models, unless loads are non-blocking in the processor. These non-blocking loads must either be scheduled statically by the user/compiler or dynamically by dynamic instruction-scheduling mechanisms. Static scheduling of loads is difficult because of intra-processor dependences. A study by Gharachorloo *et al.* [18] has shown that the performance potential attained by dynamic load scheduling does not justify the increased complexity of the processors. Therefore, in this study, we only consider standard processors with blocking loads.

2.2 Write-Invalidate Versus Write-Update Protocols

Most implementations of and proposals for large-scale multiprocessors use write-invalidate protocols [3, 21, 25] because early studies based on bus-based multiprocessors such as [13] indicated that they exhibit reduced traffic and overall better performance. Write-invalidate protocols have lower traffic because in a sequence of writes from the same processor with no intervening access from other processors, only the first write causes global traffic. Unfortunately, the invalidation of remote copies leads to coherence misses. These misses can incur a significant read penalty because the read request must be forwarded from the memory module to the cache keeping the exclusive copy. The cache with the exclusive copy then must update the memory and the block must be supplied to the requesting cache. Thus, the total read penalty to service a coherence miss includes several node-to-node block transfers.

By contrast, in write-update protocols all coherence misses are eliminated since all copies of a memory block are updated with the new value instead of invalidated on a write request to a shared block. The price to pay for the elimination of the coherence misses is an increased number of global write actions.

The write penalty under sequential consistency is substantially higher for write-update protocols than for write-invalidate protocols because of two reasons: (i) each write action incurs more latency to guarantee causal correctness [28] and (ii) the number of global write actions is larger than under write-invalidate. To guarantee causal correctness, all writes must appear in the same order with respect to each processor. Wilson and LaRowe presented a two-phase protocol [34] which guarantees causal correctness. In their protocol, two transactions per update must take place. During the first transaction, the data values are updated but the copies are locked. A processor that accesses a locked block is stalled. During the second transaction, the copies are unlocked and the processors are allowed to access their copies again. Because of the large overhead associated with this two-phase protocol, write-update protocols are not feasible in the context of sequential consistency models.

By contrast, under a relaxed consistency model, such as WO or RC, the write latency can be completely hidden provided a sufficiently aggressive design of the processor node and the memory subsystem. In addition, since causal correctness is not required for ordinary loads and stores, the two-phase update transaction is not needed. However, the potential increase in traffic can lead to more read penalty because of increased contention which overall end up increasing the execution time as compared to write-invalidate protocols. The question then is whether the traffic of update-based protocols can be kept at an acceptable level so that the reduction of read penalty they provide is not eliminated or — even reversed — by contention.

2.3 Competitive-Update Protocols

In a write-update protocol, a block loaded into a cache stays there until it is replaced, which results in update actions from other processors even if the local processor does not access the block again. To remedy this problem the local copy should be invalidated if it has been updated by remote processors a certain number of times with no intervening local access. Karlin *et al.* introduced this mechanism which they call *competitive snooping* in [22], in the context of bus-based systems, where each processing node snoops on the bus for write actions. The implementation requires a counter per cache block. On a processor access to the block, the counter is initialized to a given value, the *competitive threshold*. Whenever an update message from a remote processor is received, the counter is decremented. When the counter reaches zero, the block is invalidated.

To study the potential of read-penalty reduction of update-based protocols, we will consider a similar protocol, referred to as *competitive-update*. The competitive snooping protocol can easily be adapted to a directory-based protocol. When a cache receives an update and the counter of the

block reaches zero, the copy is invalidated and the cache notifies the memory controller of the invalidation so that updates to that cache ceases.

2.4 Summary

In summary, write-invalidate protocols have lower write traffic and write penalty at the cost of a higher coherence miss rate whereas write-update protocols eliminate coherence misses at the cost of increased write traffic in the network. Simple competitive algorithms added to a write-update protocol have the potential for reduced read penalty as compared to write-invalidate protocols while maintaining an acceptable traffic level.

Whereas the performance issues related to the policies that we consider in this paper are important, the complexity and cost of the mechanisms needed to overlap write requests with each other are critical issues. In the next section, we describe possible implementations for different write-latency hiding mechanisms which we have considered in our study.

3 Processor Node Architectures for Latency Hiding

As a basic assumption for our analysis, we have only considered design alternatives that are applicable to standard microprocessors which block on load requests. Future standard microprocessors will have an on-chip cache for data and instructions as contemporary microprocessors have. We assume throughout the paper that this on-chip cache is a write-through cache with no allocation on write misses and is blocking on read misses. Such a cache is simpler and faster than a write-back cache and is more likely to support processors with increasing clock rates. We also assume that the microprocessor has an invalidation pin and a mechanism to invalidate blocks in the on-chip cache. This pin will be required for maintaining coherence.

3.1 The Basic Processor Node

A simple and common way to hide write latencies is the cache hierarchy shown in Figure 1 which consists of a First-Level write-through Cache (FLC) with no allocation on a write miss and a Second-Level write-back Cache (SLC). To avoid processor stalls on writes, a First-Level Write Buffer (FLWB) between the two caches holds the contents of all modified words that have not updated the SLC; the processor executes all writes in one cycle in the FLWB for as long as the buffer is not backed up. Since the SLC is only accessed on a read miss from the FLC, there is plenty of SLC bandwidth to satisfy the writes in between two read misses.

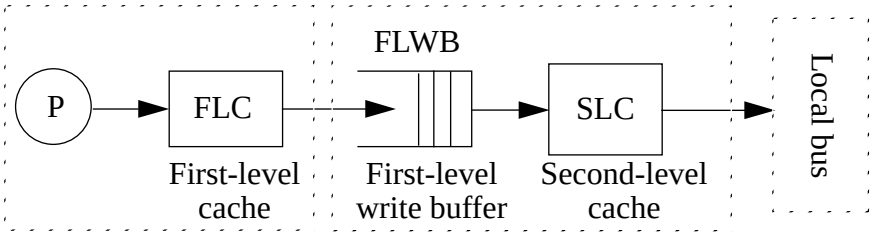

Figure 1: A basic two-level cache hierarchy.

The cache hierarchy in Figure 1 can partly take advantage of Weakly Ordered (WO) or Release Consistent (RC) memory consistency models because the processor is not blocked when a write misses or hits on a clean copy in the second-level cache: writes are buffered in the FLWB and read misses in the FLC can bypass the writes in the buffer as long as intra-processor dependences are respected, which only leads to the following three restrictions. First, synchronization operations cannot bypass the write buffer. Second, reads cannot bypass synchronization operations. Third, a read miss to a block cannot bypass a write to one of its words in the buffer. Ideally, if the read accesses a word in the write buffer, the read miss could return the value to the processor. However this would complicate the design of the buffer and the interface to the microprocessor, which might receive either a block or the word on a cache miss. For codes with read-after-write dependencies such as recurrence relations, such a mechanism may improve performance. The benchmarks we have run do not have such recurrences and therefore would not benefit from the added complexity. For these reasons, we do not consider this possibility.

Under Release Consistency, releases can be buffered with the writes in the same write buffer provided all writes preceding a release in the FIFO order of the buffer are performed before the release is issued from that buffer. Subsequent acquires and FLC read misses may bypass the releases in the buffer.

3.2 A Processor Node with a Second-Level Write Buffer

The processor node of Figure 1 is very similar to the architecture of the SGI cluster of the DASH prototype [25]. In this architecture, the second-level cache stops accepting write or read miss requests when a write emerges from the FLWB and the block is not owned in the SLC. This is a severe limitation. If we want to make the processor node truly tolerant to write latencies, we need to make the SLC lockup-free [23], meaning that it can allow multiple outstanding write requests. However, for microprocessors that block on loads, only a single outstanding read request needs to be supported. By including a second-level write buffer (SLWB), as shown in Figure 2, the SLC can allow multiple outstanding write requests. All writes that cause global actions (including write misses, invalidations, or updates) can be inserted temporarily in this buffer. Read misses in the SLC may be allowed to bypass the writes in the SLWB, under the same restrictions as for the FLWB. In contrast to the FLWB, which must be as fast as the FLC, the SLWB can afford more complex mechanisms because it is interfaced to the slower SLC. There are many design options for the architecture of the SLWB and of its controller, and we will cover some of the most important ones in this paper.

3.2.1 Design Alternatives for the First-Level Write Buffer

The first-level write buffer is needed because processor writes must complete at the speed of the processor. The major requirement of this buffer is that it must be simple and fast. Traditionally its size has been small: depending on memory latencies, 4 to 16 entries are sufficient to avoid any stall on writes in the processor.

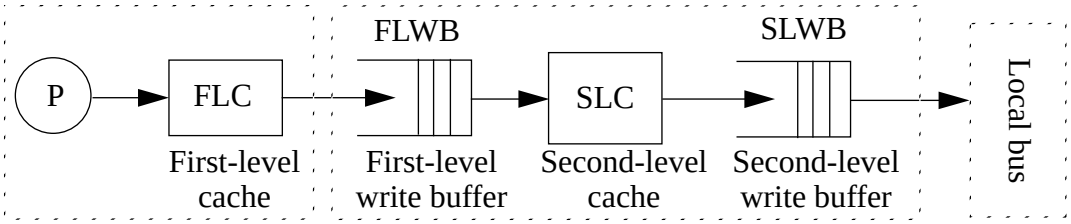

Figure 2: A two-level cache hierarchy with two separate write buffers and a lockup-free second-level cache.

One interesting design issue is whether read misses in the FLC should bypass the writes in the FLWB. Since the on-chip cache blocks on a read miss, only one such miss will ever need to bypass the buffer but the timing of the miss is critical to performance. To allow a read miss to bypass the FLWB, a mechanism must check for words of the block in the write buffer; if there is a match the read miss request is blocked until the buffer is empty.

In the case of Weak Ordering, the FLWB must be emptied at the execution of each synchronization instruction. By contrast, for Release Consistency, acquires are always allowed to bypass the write buffer. We will investigate the effectiveness of allowing read and acquire requests to bypass write and release requests in the FLWB.

3.2.2 Design Alternatives for the Second-Level Write Buffer (Write-Invalidate)

The SLWB contains entries for writes that cause global actions (misses or invalidations). It can be organized as a FIFO queue containing addresses and values of modified words. The interface to the memory system is relatively simple. If a write request is put in the buffer, the SLWB controller issues a request for ownership to the memory controller. The memory controller must deal with each request one by one. (It can queue them or it can reject them.) Based on the state of the block in other caches, the memory controller decides whether the requester needs a fresh copy of the block, which happens if the block is dirty in another cache. Note that, even if the cache had a valid copy of the block when the write was buffered, an invalidation may have removed the copy by the time the write emerges out of the SLWB. If there is a valid (non-owned) block copy in the cache and a write to the block is pending in the SLWB, writes to the block update the SLC and read misses from the FLC return the modified copy in the SLC; all updates to such blocks are also inserted in the SLWB but no additional global actions are taken. We distinguish between the following cases:

1. The block was missing in the SLC when the write was buffered. In this case, a block copy is returned by the memory system; the updates in the SLWB must be merged into the block copy before the block becomes accessible in the cache.
2. The block was not missing in the SLC but the cache did not have ownership when the write was buffered. There are two possibilities:

2.1. The block has been invalidated since the write was buffered. The local updates to the block are in the SLWB. A block copy is returned by the memory system and the updates in the SLWB must be merged into the block copy before it becomes accessible in the cache.

2.2. The block has not been invalidated since the write was buffered. The memory system gives ownership rights to the cache by returning a positive acknowledgment. All updates to the block must be removed from the SLWB when this acknowledgment is received.

A first question is whether the SLWB could issue more than one ownership request at a time. In this case, the buffer controller must keep track of each issued request until it receives an acknowledgment or a block copy; moreover, when a block copy is returned by the memory controller, all entries for the block must be removed from the buffer and possibly merged into the block copy.

A second question is the effect of read misses in the SLC. We assume that the SLC can only accept one read miss at a time (since the processor and the FLC are blocked anyway). The problems associated with allowing read miss requests to bypass writes in the SLWB are very similar as for the FLWB. If there is an entry in the SLWB for a word in the missing block, the read miss is blocked until the block copy or ownership is received, and then the miss is retried in the SLC.

Finally, under RC, releases may be buffered in the SLWB as well. Whereas the buffer may have multiple ownership requests pending, a release is not allowed to issue from the buffer until the memory responses for all the entries preceding the release in the FIFO order of the buffer have been received. Acquires may also bypass the SLWB under the same restrictions as for the FLWB.

3.2.3 Design Alternatives for the Second-Level Write Buffer (Write-Update)

Most of the design issues for the SLWB under write-invalidate also apply to a system using a write-update policy. For example, if a write misses in the SLC, we must keep track of all modified words in the SLWB and later merge them with the fresh copy from the memory. Also, we can update the SLC on a write without awaiting the pending write's completion. However, there are two differences. First, under write-invalidate a block in the SLC can be invalidated while a write to that block is in the SLWB but this cannot happen under write-update. Second, while only a single write request per block is issued from the SLWB at the same time under write-invalidate, write-update allows an unlimited number of issued writes (updates) at a time, given FIFO order of requests between two nodes in the system. Therefore, in a write-invalidate protocol, all SLWB entries to the same block can be de-allocated when the invalidation request has been globally performed, but, under write-update, we can only de-allocate the entry containing the write request which has been globally performed. Write-update protocols are likely to have a larger number of issued writes and the size of the SLWB is expected to be larger in order to fully hide the write latency. In turn, it becomes more advantageous to allow read and acquire requests to bypass

writes and releases in the FLWB when the SLWB becomes full. Note that, in order to maintain coherence, the copy of a block in the FLC is invalidated upon receiving an update request from a remote processor to a block residing in both the FLC and the SLC.

3.2.4 Design Alternatives for the Second-Level Write Buffer (Competitive-Update)

In order to keep the number of shared copies under control in an update-based protocol, a simple competitive-update mechanism consisting of a counter per SLC line keeps track of the number of updates by remote processors to a block. On a read miss in the FLC, the block is loaded into the FLC and the counter associated with the block in the SLC is initialized to a predefined value C, the *competitive threshold*. When the SLC receives an update message from another processor node, the actions taken depend on the value of the counter.

1. If the counter is not zero, it is decremented, the corresponding block in the SLC is updated, and the block in the FLC is invalidated. In addition, an UpAck (update-acknowledgment) message is returned to the memory controller.
2. If the counter is zero, the blocks in the SLC and FLC are invalidated and an UpAckInv (update-acknowledgment-invalidated) is returned to the memory controller indicating that the processor node does not have a copy any longer.

Consequently, after C consecutive update messages to a block from other nodes with no intervening local reference to the block, the block is invalidated. Like write-invalidate protocols, the block becomes exclusive (dirty) in an SLC if no other SLC has a copy of the block. With this implementation, we manage to keep the FLC fast and simple. The only two external operations on the FLC are invalidation of a block and loading a block. All complexity of the competitive mechanism is handled in the SLC.

The competitive threshold is an important design parameter, which we will study later in this paper. Too small a threshold prevents the protocol from reducing the coherence miss rate and thus the read penalty, whereas too big a threshold generates excessive write traffic, which may impact adversely on the read-penalty reduction.

4 Simulation Methodology, Architectural Designs, and Benchmark Programs

In this section, we present the evaluation methodology, including the simulation environment and the detailed architectural assumptions (Section 4.1), the buffering alternatives for the cache hierarchy (Section 4.2), and the benchmark programs (Section 4.3).

4.1 Simulation Environment and Memory System Assumptions

The simulation models are built on top of the CacheMire Test Bench [7], a program-driven simulator and a programming environment. The simulator consists of two parts: (i) a functional simulator of multiple SPARC processors and (ii) an architectural simulator. The SPARC processors in the functional simulator issue memory references and the architectural simulator delays the pro-

cessors according to its timing model. Thus, a correct interleaving of memory references is maintained by keeping track of the global time because the sequence of memory references is derived by correctly modeling the delays in the target architecture.

The high-level organization of the processor node model we study is shown in Figure 3. The two-level cache hierarchy we simulate consists of a 2 Kbyte FLC and an infinite SLC, both with a cache-line size of 16 bytes. While we also study variations on the size of the write buffers, we will assume that they both contain 16 entries by default. The cache hierarchy, referred to as the processor environment, is interfaced to the local portion of the shared memory and the Network Interface Control (NIC) by a local bus according to Figure 3.

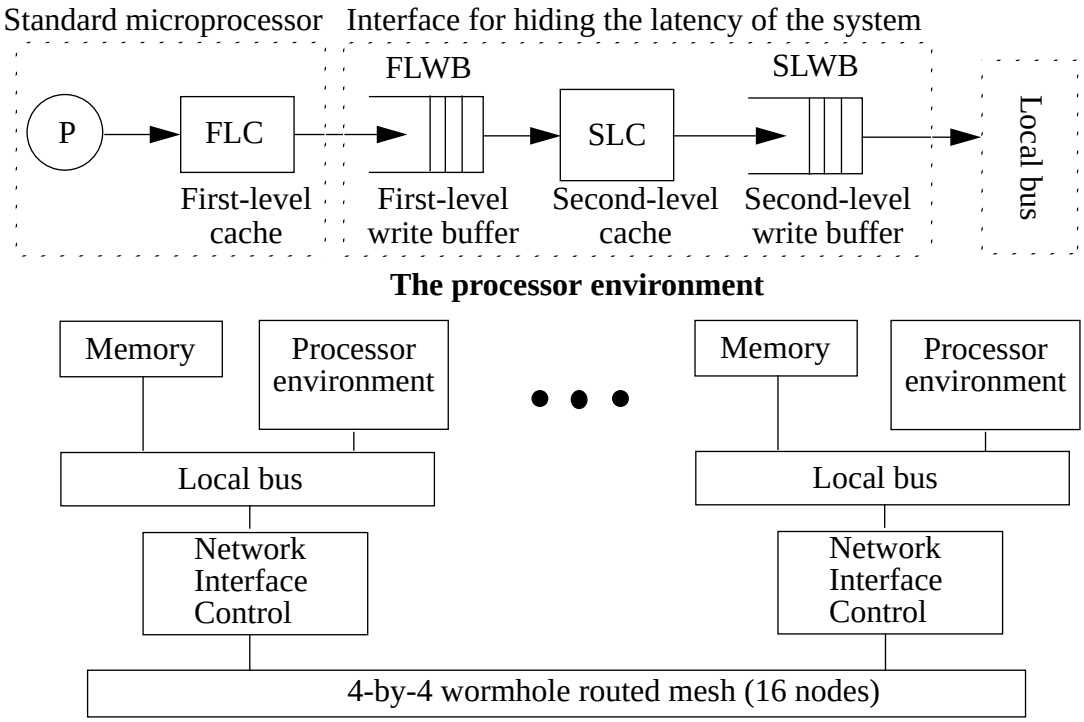

Figure 3: The processor environment and the simulated architecture.

Cache coherence is supported by a directory-based protocol similar to Censier and Feautrier's [9]; each memory block is associated with a directory entry containing a presence flag vector indicating which nodes have a copy of the block. The memory module in which a particular block is allocated is called the *home* of that block. The page size is 4 Kbyte and we assume that pages are allocated to memory modules in a round-robin fashion; pages are interleaved across nodes. A read miss in the SLC sends a read request to home. If the home is the local node and if the memory block is clean, the miss is serviced locally. Otherwise, the miss is serviced either in two or in four node-to-node traversals depending on whether the block is clean or dirty. Upon an invalidation or an update request, the home memory controller is responsible for sending explicit invalidations or updates to each node according to the state of the presence flag vector and the global coherence state of the block. An invalidation/update from the memory module generates one single message

on the local bus, including a presence flag vector; then the NIC sends explicit messages to each node with a copy of the block. The NIC is also responsible for collecting invalidation/update acknowledgments from other nodes. When the NIC has collected all acknowledgments, it sends a single acknowledgment over the bus to the memory controller. Finally, acquire and release requests are supported by a queue-based lock mechanism similar to the one implemented in DASH [25]. A block where a lock variable is allocated contains only that single lock variable and no other variables.

As far as the timing and architectural parameters are concerned, we consider SPARC processors and FLCs that are clocked at 100 MHz (1 pclock = 10 ns). The access time of the SLC is assumed to be 30 ns (SRAM technology). The memory is assumed to be built from DRAM technology and is fully interleaved with an access time of 90 ns. The SLC and its write buffer are connected to the NIC and the local memory module by a 128-bit wide split transaction bus clocked at 100 MHz. Thus, it takes 10 ns to arbitrate for the bus and 10 ns to transfer a request or a block. We simulate a very fast bus since the bus load is an orthogonal issue in this study. Table 1 shows the time it takes to service a read request when data is fetched from different levels in the memory hierarchy, assuming a 100 MHz processor and a contention-free system. (In our simulations, requests will normally take a longer time as a result of contention.) We simulate a system with 16 nodes interconnected by a 4-by-4 wormhole routed synchronous mesh with a flit size of 64 bits and clocked at 100 MHz to be compatible in speed with the processors. The aggregate bandwidth out from and into each node is 800 Mbytes/second. We model contention for all components in the processor node and in the mesh network.

Table 1: Latency of processor read requests when data is supplied from different levels in the memory hierarchy.

Latency of read requests	1 pclock=10 ns (100 MHz)
Fill from FLC	1 pclock
Fill from SLC	4 pclocks
Fill from Local Memory	20 pclocks
Fill from Home (2-hop)	43 pclocks
Fill from Remote (4-hop)	82 pclocks

4.2 Restrictions on Buffering

We have simulated three different design alternatives for the cache hierarchy under Release Consistency. These designs differ in the aggressiveness of the lockup-free mechanism of the SLC. In addition, we also evaluate the effectiveness of read misses bypassing writes in the FLWB in the context of each model.

Model RC-I corresponds to an architecture with a FLWB between the first-level and second-level caches as shown in Figure 1. The SLC is blocking and there is no SLWB; a write request causing a global action (write miss, invalidation, or update) blocks the SLC for as long as the request is pending. Note that the processor is blocked only if the FLWB is full or if a read access misses in the FLC.

In models RC-II and RC-III, global write requests are buffered in the SLWB. Under model RC-II and RC-III we evaluate read miss and acquire bypass in the SLWB, but not in the FLWB. There can be only one pending acquire or read miss request in the SLC. The only difference between model RC-II and model RC-III stems from the number of pending write requests in the network. While model RC-II allows only one pending read and one pending write request issued from the SLWB at a time, model RC-III allows as many pending write requests as there are entries in the SLWB, with the restriction that no more than one request to the same block is issued to the network in the write-invalidate protocol at any time. Table 2 summarizes the features of the design alternatives.

Table 2: Simulated architectural models.

Architectural model	Second-level cache (SLC)	Second-level write buffer (SLWB)
RC-I	Blocking. One pending request at a time. The cache is blocked until the global request is performed.	None
RC-II	Lockup-free. The cache is blocked only when the SLWB is full.	One pending read and one pending write request at a time. Read misses bypass the buffer if no write to the same block is in the buffer. Releases are buffered and acquires always bypass the buffer.
RC-III	Lockup-free. The cache is blocked only when the SLWB is full.	One pending read and as many pending write requests as entries in the buffer. Read misses bypass the buffer if no write to the same block is in the buffer. Releases are buffered and acquires always bypass the buffer.

Under WO and RC reads are allowed to bypass writes in the write buffers, as long as they are not to the same address, and thus we also evaluate the effectiveness of a read-bypass mechanism added to the FLWB of each model. Since acquires are treated as read requests to synchronization variables, the processor blocks on acquires just like it does on read misses. Under RC, which is the default memory consistency model, acquires are allowed to bypass previous writes and releases. When the three models are extended with a bypassing mechanism in the FLWB, we refer to them as RC-I-bp, RC-II-bp, and RC-III-bp.

4.3 Benchmark Programs

In order to understand the relative performance of our architectural models under various coherence policies, we use four scientific and engineering applications, all taken from the SPLASH suite [29] except for the C-version of Ocean which has been provided to us by Steven Woo at Stanford University. The main characteristics of the four benchmark programs, MP3D, Water, PTHOR, and OCEAN, together with the size of the data set used are summarized in Table 3. All programs are written in C using the PARMACS macros from Argonne National Laboratory [6] and compiled with `gcc` version 2.1 (optimization level -O2). Statistics are collected during the execution of the parallel section in the applications.

Table 3: Benchmark programs.

Benchmark	Description	Data sets/Input
MP3D	Particle-based wind-tunnel simulator	10 000 particles, 10 time steps
Water	Water molecular dynamics simulation	288 molecules, 4 time steps
PTHOR	Simulation of a digital circuit at the logic level	RISC circuit, 1000 time steps
Ocean	Simulation of eddy currents in an ocean basin	128-by-128 grid, tolerance 10^{-7}

5 Experimental Results

We start by comparing the performance of the three coherence policies in Section 5.1. In Section 5.2 we compare the performance of different buffering schemes. The impact of the competitive threshold (Section 5.3) and of the consistency models (Section 5.4) follows. Finally, in Section 5.5, simulation results for different network speeds are given in order to see how sensitive our qualitative conclusions are to network capacity.

5.1 Relative Performance of Write-Invalidate, Competitive-Update and Write-Update

In order to separate implementation issues from the performance gains of the various coherence policies, we analyze the performance of write-invalidate, competitive-update, and write-update by assuming an aggressive lockup-free second-level cache according to model RC-III in Section 4.2 with 16 entries in the second-level write buffer.

The execution times for the applications are found in Figure 4. All execution times are normalized to the execution time for write-invalidate for each application. The different sections in the bars correspond to (bottom to top): the busy time (or processor utilization), the processor stall time due to read misses, the processor stall time to perform acquire requests, and the processor stall time due to a full first-level write buffer. The three bars for each application correspond to the three coherence policies: write-invalidate (W-I), competitive-update (C-U), and write-update (W-U). In our measurements, we have assumed a competitive threshold of 4.

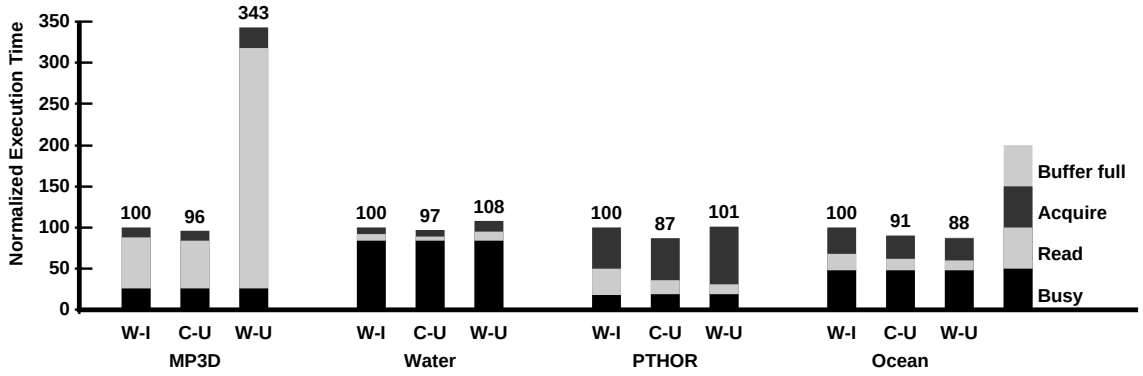

Figure 4: Normalized execution time of the benchmarks for the three coherence policies under RC.

Let us first compare write-invalidate with write-update. In Figure 4 we see that write-invalidate results in significantly better system performance than pure write-update for two of the applications (MP3D and Water). While the stall time due to acquires and full buffers is about the same, the reason for the performance difference is the longer read stall time under write-update. The reason why the read stall time is increased despite of the elimination of coherence misses is because of contention due to the increase in network traffic as a result of the updates. To see this, we also measured the network traffic for all applications under write-invalidate and write-update. This data appears in Figure 5.

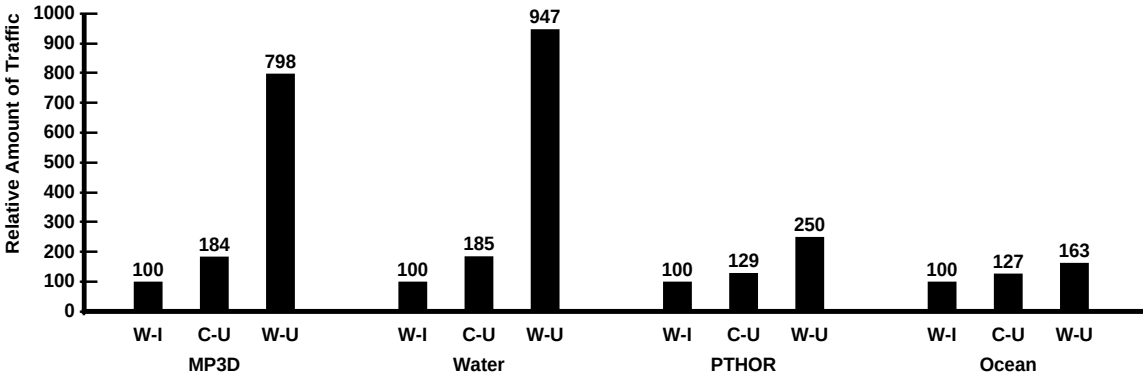

Figure 5: Relative amount of network traffic generated under the different coherence policies.

Figure 5 shows the amount of network traffic for write-update (W-U) relative to write-invalidate (W-I). The traffic is measured in number of flits sent through the network and is normalized to the traffic rate under the write-invalidate protocol for each application. The traffic under write-update is 7 to 10 times more than under write-invalidate for MP3D and Water. For Ocean, write-update performs significantly better and the traffic level is acceptable. MP3D and Water have poor performance under write-update because of migratory sharing [20, 32]; as a migratory block migrates from cache to cache, it creates copies that may not be referenced for a long time, flooding the network with updates. By contrast, Ocean is based on an iterative algorithm and values are communicated among neighboring processes. Therefore, write-update performs much better than write-invalidate.

We now turn our attention to competitive-update protocols. The main objectives of competitive-update protocols is to reduce the network traffic generated by write-update protocols and at the same time to take advantage of the elimination of most coherence misses. In Figure 5 we see that the competitive-update protocol successfully reduces the network traffic by up to 80%, as compared to the write-update protocol. As compared to write-invalidate, competitive-update generates about 85% more traffic for MP3D and Water which are applications exhibiting migratory sharing, but only about 30% more for the other two applications. This does not seem to be a critical issue since the network is capable of handling that extra amount of traffic effectively.

In Figure 6 we show the relative read penalty under the different coherence policies. We observe that the competitive-update protocol successfully reduces the read penalty (from 6% to 46%) compared to the write-invalidate protocol **for all applications**. The write-update protocol can reduce the read penalty even further for applications with little migratory sharing (PTHOR and Ocean). Thus, competitive-update is a better default policy than write-invalidate for all four applications.

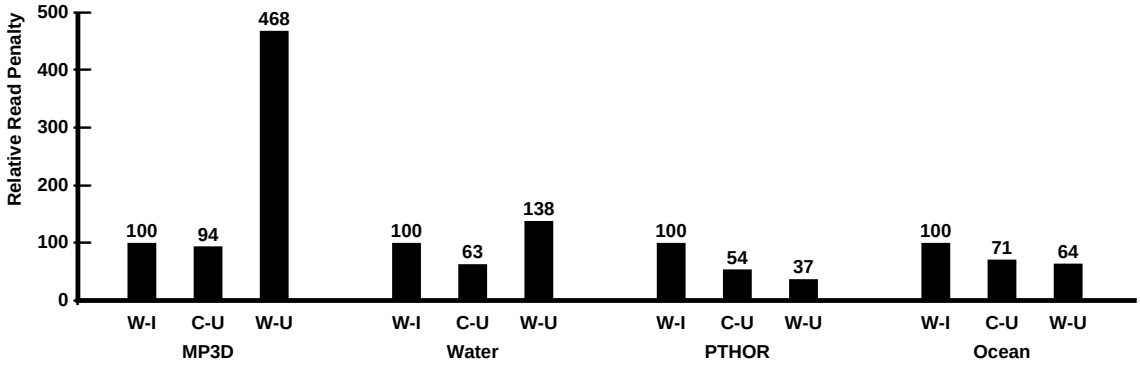

Figure 6: Relative read penalty under the different coherence policies.

We now look at the two factors affecting the read penalty under competitive-update. The first factor is the reduction of the coherence miss rate and the second factor is the fraction of times a block is clean at the memory on a read miss, i.e., no cache has an exclusive copy of the block. The effect of the first factor is clear, but some increase in network traffic is needed to update the cached copies, which in turn may increase the latency time of a cache miss. The second factor has a positive effect because if the block is kept up-to-date at the memory, a read miss to the block costs at most 2 network traversals in contrast to at most 4 network traversals if the block is exclusive (dirty) in another cache. The effects of the two factors are summarized in Table 4.

We see in Table 4 (right column) that competitive-update reduces the read latency for all applications except MP3D. For example, the average time for a read request to complete in Ocean is 75 pclocks under competitive-update whereas the corresponding number for write-invalidate is 87 pclocks. We also observe that a block rarely becomes dirty under competitive-update as compared to write-invalidate (the middle column); as many as 100% - 16% = 84% of all misses to blocks that are dirty under write-invalidate for Ocean can be serviced at the memory module

Table 4: Statistics for read misses in the SLC for competitive-update relative to write-invalidate.

Application	Relative coherence miss rate		Relative numbers of read misses to dirty blocks		Time for a read miss request to complete (pclocks)	
	W-I	C-U	W-I	C-U	W-I	C-U
MP3D	100%	87%	100%	5%	95	92
Water	100%	80%	100%	6%	80	51
PTHOR	100%	34%	100%	19%	114	81
Ocean	100%	24%	100%	16%	87	75

under competitive-update. Since the competitive threshold is set to 4, i.e., a block becomes exclusive in a cache if a processor writes 4 times to the block with no other processor accessing it, we conclude that most data blocks are read and modified by different processors in an interleaved fashion. There is a clear distinction in the reduction of coherence misses between PTHOR and Ocean on one hand, and MP3D and Water on the other hand. For PTHOR and Ocean the coherence misses are reduced by about two thirds, but for MP3D and Water the reduction is only 13% and 20%, respectively. The difference can be explained by observing that most data objects in MP3D and Water are migratory objects [20, 32]. For MP3D the time for a read request to complete decreases very little under competitive-update as compared to write-invalidate; even though the blocks are clean in memory for 95% of the misses, the network contention induced by increased write traffic offsets the reduction in the pure miss latency. However, the reduced coherence miss rate under competitive-update cuts the overall read penalty by 6% for MP3D as compared to write-invalidate (see Figure 6).

An overall reduction in the execution time of 3 to 13% is observed in Figure 4. A negative effect of competitive-update, however, appears in the case of PTHOR. Namely, the acquire stall time is higher under competitive-update and under write-update than under write-invalidate. A release residing in the write buffer can not be issued from the processing node until all previous writes are performed. If another processor waits for the release, it may see an increased acquire stall time due to the delayed execution of the release. The higher number of global writes under write-update leads to an increased acquire stall time. As a result, for applications exhibiting contention for critical sections (or locks) the write latency may be converted into increased synchronization overhead.

In summary, competitive-update protocols successfully reduce the coherence miss rate and the read stall time, which results in shorter execution times under competitive-update than under write-invalidate for all applications. We have also seen that competitive-update maintains an acceptable traffic level as compared to write-invalidate. However, update-based protocols may increase synchronization overhead for applications that exhibit contention for critical sections.

5.2 Evaluation of Different Buffering Alternatives

In this section we compare the buffering alternatives described in Section 4.2 for write-invalidate and competitive-update protocols.

5.2.1 Buffering Alternatives for Write-Invalidate Protocols

We start by analyzing the impact of the three buffering alternatives on the performance under the write-invalidate policy. The results are compared with the performance of Sequential Consistency (SC). Our implementation of SC forces the processor to stall on each shared data access. The write buffers have a limited size; each buffer is 16 entries deep. Figure 7 shows the execution times for the benchmark applications under write-invalidate.

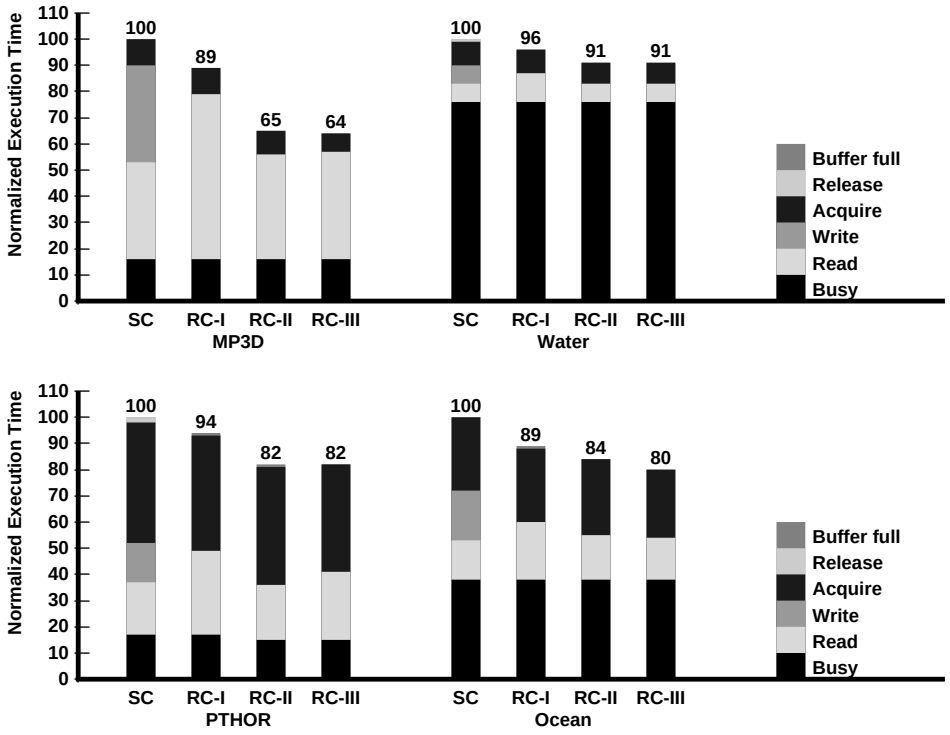

Figure 7: Normalized execution time of the benchmark applications for the buffering alternatives under write-invalidate.

In our first model (RC-I) buffering is limited to a FLWB with no read bypass. We observe reductions of the execution times by 4% to 11% as compared to SC (Figure 7). The only write latency we can hide is the time from a write access to a following read miss in the FLC. The read request has to wait until the write is completed, which may be the time it takes to perform the write globally if the distance between the write and the subsequent read miss is short. During that time the SLC is blocked and can not handle the read request. Therefore, most of the write latency from SC is converted into read latency in the RC-I model. The only exception is Ocean, where less than 40% of the write latency is converted into read latency. This result indicates that there is a longer distance between a global write access to a following read miss in the FLC in Ocean than in the other three applications. From Table 5 we see that a read request spends only 3 pclocks in

the FLWB for Ocean, as opposed to 42, 12, and 22 pclocks for MP3D, Water, and PTHOR, respectively. MP3D is the only application where there is more than one request in the buffer when the read miss occurs. This indicates that MP3D is the only application where read bypassing in the FLWB has a potential to reduce the read penalty.

Table 5: Queuing statistics for an FLC read miss in the FLWB under buffering model RC-I.

Application	Average time in the FLWB (pclocks)	Average number of requests before a read request in the FLWB
MP3D	42	1.4
Water	12	0.5
PTHOR	22	0.1
Ocean	3	0.1

To test this intuition, we evaluated the effects of read bypassing in the FLWB for each benchmark under RC-I and did not observe any performance gain at all, except for MP3D where a small decrease in read stall time was observed. The reason is that the SLC is still blocked due to pending write requests at the time the read miss occurs. We also see in Table 5 that at the time a read request is issued to the FLWB the buffer is mostly empty. If there are multiple write requests in the FLWB requiring global actions, we would expect to see a larger performance gain from read request bypassing in the FLWB, but this is clearly not the case. For one of the applications, PTHOR, we even observed an increase in execution time by 1% for the RC-I-bp model as compared to the RC-I model. The reason is that the acquire stall time has increased as an effect of the delayed issuance of releases. From our measurements we found that the average time a release spends in the FLWB increases from 87 to 147 pclocks for PTHOR when read bypassing is allowed in the FLWB.

From Figure 7, we see that the execution times drop when one outstanding read request is issued in parallel with one outstanding write request, as in model RC-II. This requires the SLC to be lockup-free [23, 31] and a SLWB is introduced with a single pending global write request. The read latency is reduced to almost the same level as in SC for all of the applications because a read miss request can be issued from the SLC at once. For MP3D we observe a slightly higher read penalty under RC-II than under SC, which comes from increased network contention. The reduction in the total execution times with respect to RC-I comes from the reduced read stall times in all four applications. We did not see any problems in hiding the write latency for any of the applications.

As can be seen in Figure 7, moving from model RC-II to RC-III does not buy us any significant performance increase in general under write-invalidate. In model RC-III, the acquire stall time for PTHOR is lower than RC-II because releases are issued faster from the SLWB when

multiple pending write requests (up to 16) are allowed. PTHOR is an application with a high rate of synchronizations and it benefits from the fact that releases are issued faster from the processing nodes. To take advantage of Release Consistency as much as possible in the general case, multiple pending write requests are necessary. Unfortunately, in PTHOR we observe that the reduced acquire stall time is converted into read latency because of network contention so the overall performance increase is negligible.

We also evaluated the performance gains for RC-II and RC-III when read requests are allowed to bypass write requests in the FLWB, leading to models RC-II-bp and RC-III-bp. We did not see any performance benefit for the same reasons as for RC-I-bp; the SLWB is large enough, so there is never any request in the FLWB when the read request is issued from the FLC. Moreover, the SLC is not blocked by previous requests either.

In summary, we observe the main performance increase when the SLC is lockup-free and a SLWB is present so that one read miss request can be issued in parallel with one global write request. Supporting multiple pending global write requests does not yield any significant performance improvement under write-invalidate. Allowing read requests to bypass write requests in the FLWB does not yield any significant improvement either since the FLWB is mostly empty at the time it receives a read request.

5.2.2 Buffering Alternatives for Competitive-Update Protocols

In this section we compare the different buffering alternatives under competitive-update. The results are summarized in Figure 8. The baseline model is an implementation where the processor is stalled at each shared read or write request and referred to as SC although it does not maintain Sequential Consistency in a strict sense. The update transactions are performed in a single pass, not in two as discussed in Section 2.2.

When we go from SC to RC-I under competitive-update, we observe the same phenomenon as for write-invalidate; a part of the write latency is converted into read latency. The amount of write latency converted differs among applications. For MP3D, Water, and PTHOR almost all the write latency is converted into read latency, whereas for Ocean only about two thirds of the write latency is converted.

Introducing a SLWB in the cache hierarchy with one pending read and one pending write request, as in model RC-II, yields a significant performance increase under competitive-update. The execution time is reduced by 11% to 19% as compared to RC-I, mainly due shorter read stall time. Since one read and one write request can be outstanding at the same time, the write request does not delay the issuance of a read request as in model RC-I. There is a small increase in the acquire stall time for Ocean, however, due to contention effects in the network which delay global write requests and releases.

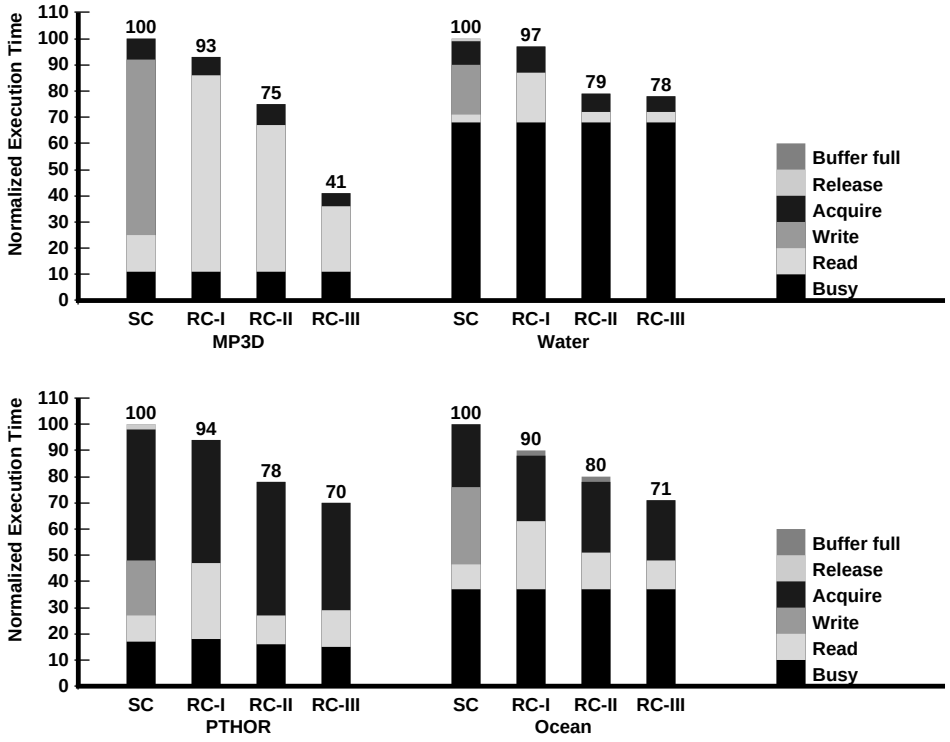

Figure 8: Normalized execution time of the benchmark applications for the buffering alternatives under competitive-update.

In contrast to write-invalidate, we observe from the results in Figure 8 that allowing the issuance of multiple write requests from the SLWB further reduces the execution times for all applications. In MP3D this stems from the reduced time a read request spends in the FLWB. Global write requests are retired from the SLWB at a higher rate in model RC-III than in model RC-II which makes the SLC service the requests from the FLWB at a higher speed. In PTHOR and Ocean the reduced execution times mainly stem from a reduced acquire stall time. Since global writes are completed faster, releases residing in the write buffers can be issued from the processing node faster and, as a result, acquires can complete faster if they are waiting for the release. Thus, we conclude that it is essential to allow multiple pending writes in order to benefit from the performance potential of the competitive-update protocol. By using a relaxed memory consistency model and appropriate hardware support, the execution times for the applications are reduced by between 22% and 59% as compared to our SC model.

When we allowed read requests to bypass the FLWB, as in model RC-I-bp, we did not see any significant improvement as compared to RC-I, except for Water where a reduction of the execution time is due to an almost 50% shorter read stall time (not shown in Figure 8). For Water, a read request that bypasses writes in the FLWB in model RC-I-bp, bypasses 3 write requests on the average. As a result, the write requests and releases are delayed in the FLWB. However, since the locks are not contended in Water, all write latency can be hidden. For MP3D and Ocean, we observed a significant decrease of the read stall time, but the processor stall time due to a full first-

level write buffer was increased by the same amount. For example, in MP3D each read miss bypasses 12 writes in the FLWB on the average. This causes the FLWB to be filled, and as a result, the processor is stalled.

We have also evaluated the performance gain when read requests are allowed to bypass write requests in the FLWB in the presence of an SLWB. We did not observe any overall performance gain of read bypassing under competitive-update. We found that the read latency is slightly reduced for some applications, but the processor stall time due to a full first-level write buffer is increased by approximately the same amount as the read latency is reduced.

In summary, unlike write-invalidate, it is essential to allow multiple pending write requests under competitive-update to benefit as much as possible from the latency hiding capability of Release Consistency. Like write-invalidate, a significant performance gain is achieved with only one pending read and one pending write request at the same time for all applications; moreover read bypassing in the FLWB does not improve the performance significantly, especially when the SLC is lookup-free.

5.3 Effects of Various Competitive Thresholds

In our default competitive-update protocol we have used a competitive threshold of 4, i.e., a cached copy is invalidated when it has been updated 4 times since the last reference by the local processor. In this section we show simulation results with competitive thresholds from 1 to 8.

Figure 9 summarizes the results from the simulations with various thresholds for the competitive-update protocol. The execution times are normalized to the execution time under the write-invalidate (W-I) protocol which is the leftmost bar for each application. The next four bars to the right correspond to the execution times under competitive-update (C-U) with thresholds 1, 2, 4, and 8, respectively. The sixth bar for each application is the normalized execution time under the write-update (W-U) policy.

For MP3D we see that the execution time is almost the same for competitive thresholds of 1 and 2 as for write-invalidate. For a competitive threshold of 4 competitive-update has a shorter execution time than write-invalidate, and for higher thresholds the execution time increases again. The extreme point is for the write-update protocol, where MP3D runs almost 3.5 times slower than for the write-invalidate protocol, as a result of the intense migratory sharing leading to high communication bandwidth.

For Water, we see that a threshold of 4 or 8 results in the shortest execution time. In Water most data objects are migratory, but the communication to computation ratio is lower than in MP3D. Therefore, the mesh network is not so heavily loaded, and most of the write traffic can be hidden by local computation without affecting the read requests. In fact, the read stall time decreases as the competitive threshold increases since the coherence miss rate and read latency decrease. Also Water suffers from a huge amount of write traffic under the write-update protocol

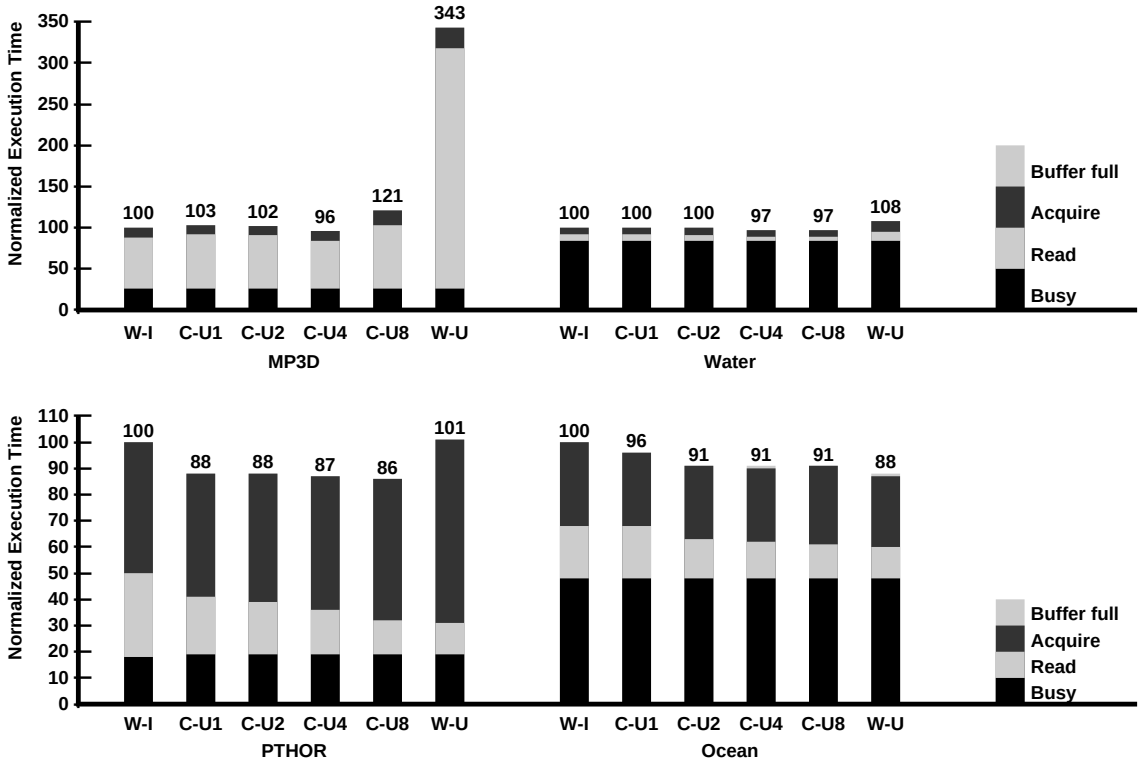

Figure 9: Normalized execution time under competitive-update with various competitive thresholds.

and an increased read penalty is observed. However, the execution time under the write-update protocol is only 8% longer than under the write-invalidate protocol.

For PTHOR the best competitive threshold is 8. The read penalty for PTHOR decreases as the competitive threshold increases, so the application benefits from keeping the caches updated. However, when the write traffic increases as the threshold increases, releases are delayed in the write buffers before they leave the processing node, which in turn increases the acquire stall time.

Ocean is the only application where the write-update protocol results in shorter execution time than the write-invalidate protocol. The execution time is reduced by 12% under write-update as compared to write-invalidate, mainly because of the reduced read stall time. We also observe that the threshold is not critical for Ocean, as long as the threshold is larger than one.

In summary, we have observed that most applications benefit from keeping the caches updated. The read penalty is reduced when the competitive threshold increases, but the acquire stall time may increase at the same time due to the increased write traffic which delays the issuance of releases. The best competitive threshold for an application is difficult, or even impossible, to predict statically. The optimal threshold varies between applications depending on the communication to computation ratio and the access patterns to shared data structures. The best threshold may vary between blocks within the same application and possibly for the same block during the execution of the program. We may say that as long as the competitive threshold is not extremely large, competitive-update protocols provide consistent performance improvement.

5.4 Weak Ordering vs. Release Consistency under Update-Based Protocols

Previous studies [17, 36] have shown that there is little or no performance difference between Weak Ordering and Release Consistency under write-invalidate. However, no previous study has addressed the relative effectiveness of the two consistency models under update-based protocols. One could argue that an application would run equally fast under WO and RC since both models have the possibility to overlap write requests with local computation and read requests. On the other hand, one could argue that the higher rate of global write actions under update-based protocols favors RC.

In order to understand which relaxed consistency model is better, we ran the four applications under both consistency models on an architecture with our most aggressive buffering alternative (model III) and 16 entries deep write buffers. We simulated both a network with infinite bandwidth and our default network; a 100 MHz mesh network with 64-bit wide links. The infinite-bandwidth network allows us to isolate the differences stemming from the consistency models and not from the implementation. Under WO, the processor must stall at a synchronization point until the write buffers are emptied. This stall time is classified as write stall time.

We first discuss the results obtained with the infinite-bandwidth network. Comparing RC and WO for each of the three policies, W-I, C-U, and W-U, we observed no significant difference in performance between RC and WO for MP3D and Ocean. However, for Water and PTHOR we found a difference between RC and WO, which is in accordance to the results for write-invalidate in [17]. Water has very short critical sections and PTHOR is the only of our four applications with a high synchronization rate. For Water and PTHOR, the lower execution times under RC than under WO (8% and 12%, respectively, under C-U) stem mainly from the increased write stall time and, to a lesser extent, from the increased time to perform release requests under WO. Simulation results also showed that the difference between RC and WO increased as we go from W-I, to C-U, and to W-U. As expected, the higher rate of global write actions directly affects the write stall time under WO.

The results from the architecture with the mesh network yielded a similar difference between RC and WO as the infinite-bandwidth network did. However, the difference is smaller as a result of network contention. For C-U, the execution times under RC are 6% and 9% lower for Water and PTHOR, respectively, than under WO. Also for the mesh network we found an increasing difference between RC and WO as we go from W-I, to C-U, and to W-U. The write stall times under WO increase with the write traffic (from W-I to C-U to W-U) and this effect is even more pronounced when contention is taken into account.

In summary, by running the four applications under both Weak Ordering and Release Consistency we have not found any difference between WO and RC for two of the applications (MP3D and Ocean) in terms of latency hiding capabilities under update-based protocols. However, for Water and PTHOR we observed a slight difference between RC and WO as a result of very small critical sections (Water) and a high synchronization rate (PTHOR).

5.5 Effects of the Speed Gap Between Processor and Network

To get a feel for the sensitivity of our results to variations in speed differences between the processors and the network, we simulated an architecture with 100 MHz processors and three different networks; (i) a network with infinite bandwidth but latencies comparable to a 100 MHz mesh, (ii) a 100 MHz mesh, and (iii) a 33 MHz mesh. The latter case corresponds to a network clocked three times slower than the processor. The execution times shown in Figure 10 are normalized to the execution time under write-invalidate.

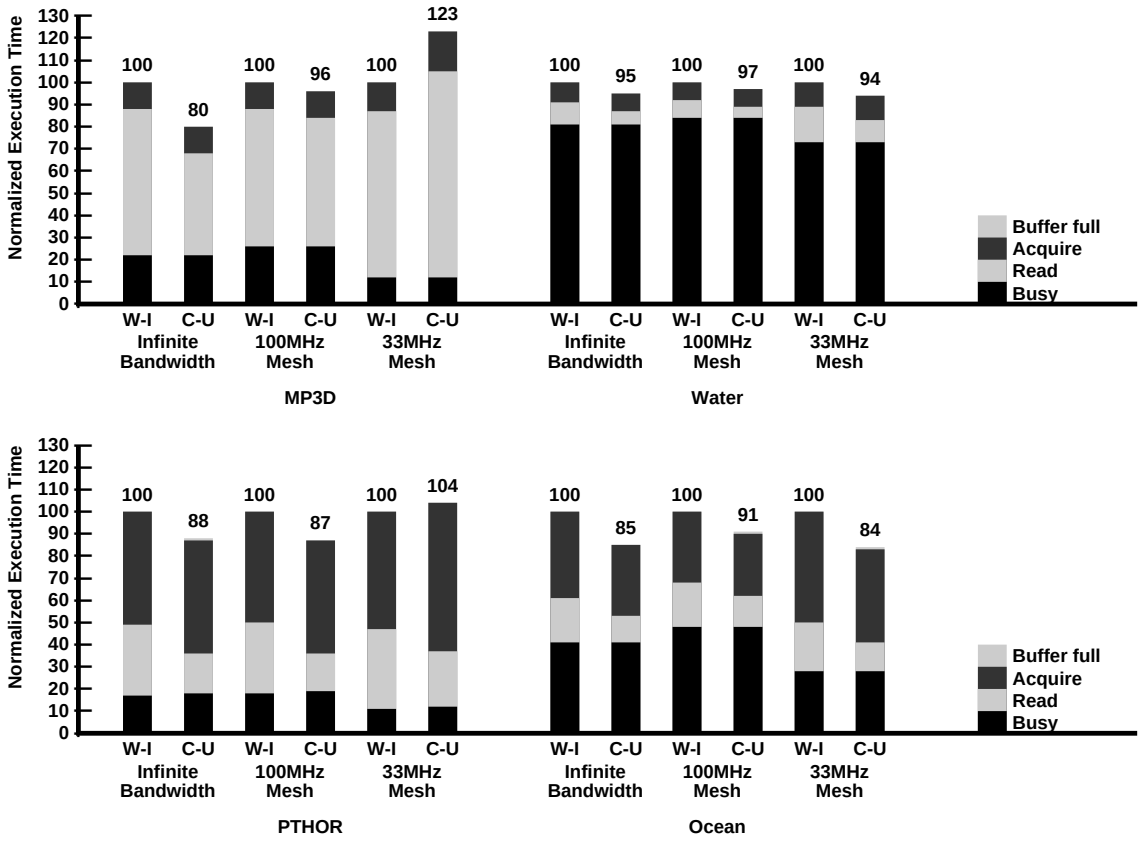

Figure 10: Effects of varying the capacity and the speed of the network

By comparing the execution times of all applications under write-invalidate (W-I) and under competitive-update (C-U) for each network, we see that the relative difference between write-invalidate and competitive-update is approximately the same for the infinite bandwidth network and the 100 MHz mesh except for MP3D, which has a very high bandwidth demand. We also see that competitive-update gives a **consistent performance improvement** over write-invalidate in the architecture with a 100 MHz mesh network. For the 33 MHz mesh network, Water and Ocean have shorter execution times under competitive-update than under write-invalidate. For MP3D the high communication-to-computation ratio and the predominance of migratory sharing result in a high rate of update messages and in severe network contention. Therefore, it is not possible to hide the write traffic as effectively as with the faster 100 MHz mesh. For PTHOR the high synchronization rate affects the total execution time. Nevertheless, competitive-update reduces the

read penalty for PTHOR even on the slower mesh, but the high rate of global write actions affects the overhead of releases and acquires adversely. From our measurements we find that competitive-update reduces the read penalty by 32% to 43% as compared to write-invalidate for all applications but MP3D on the architecture with a 33 MHz mesh.

Qualitatively our conclusion that competitive-update successfully reduces read penalty still holds even with a mesh network three times slower than the processors. In other words, the read penalty reduction of competitive-update protocols is not very sensitive to changes in network capacity given that the ratio of computation to communication and the distances between synchronizations are sufficiently large.

6 Discussion and Related Work

The performance evaluations reported in this paper show that competitive-update consistently performs better than write-invalidate for applications with moderate bandwidth requirements and small contention for critical sections. Although the idea of using hybrid update/invalidate protocols is not new, previous proposals have been specifically studied in bus-based systems and rely on snooping. As we will discuss in the following, the trade-offs become fundamentally different in a directory-based environment.

In [14], Eggers and Katz compare write-invalidate and write-update snoopy-cache protocols. They conclude that neither protocol is better than the other. Our results show that pure write-update is highly undesirable in the general case because of the heavy traffic caused by updates. Eggers and Katz also evaluate two extensions to pure write-invalidate and write-update called read-broadcast and competitive snooping [15]. While they show that competitive snooping can improve the performance of write-update, they do not compare the performance of competitive snooping with write-invalidate as we do. Eggers and Katz's competitive snooping protocol is an implementation of Karlin's Snoopy-Reading protocol [22]. A writing processor keeps track of the number of its own consecutive writes to each address. When the threshold for broadcasts has been reached, the processor sends an invalidation on the bus to the other caches. When another cache re-reads the block all caches with an invalid copy of the block catch the copy of the block as it propagates on the bus (read-broadcasting) and reset their counters to the threshold. Clearly, read-broadcast is not feasible in a directory-based environment.

In [4], Archibald proposed an adaptive write-invalidate/write-update snoopy-cache protocol. His adaptive protocol starts in update mode, just like ours, and when a single processor has issued three consecutive writes to the same block without any intervening access by another processor, all other copies of the block are invalidated. A significant difference between Archibald's and our protocol is the behavior for migratory data objects. His protocol invalidates all copies of a block when the same processor has written three times to the block, while in our protocol writes from several processors contribute to the invalidation of a block copy. As a result, for migratory data objects where each processor updates a block less than three times, Archibald's protocol will con-

tinuously update all block copies, which may degrade the performance. In contrast, our protocol will update at most three copies of the block, given the same threshold as Archibald's protocol, which significantly reduce the write traffic in a directory-based environment.

Veenstra and Fowler have evaluated the performance of optimal hybrid protocols in [33]. They use three types of hybrid protocols: Static Hybrid (each block uses W-I or W-U for the entire execution), Paged Hybrid (all blocks in a page uses either W-I or W-U for the entire execution), and Dynamic Hybrid (the protocol chooses between W-I and W-U at each write). By using off-line optimal analysis, they found that hybrid protocols may offer substantial performance advantages over W-I or W-U, especially for large block sizes. As expected, the Dynamic Hybrid protocol performs best, followed by the Static hybrid and the Paged Hybrid protocols. However, they found that using the static strategy was almost as good as the dynamic one. Their results also indicate that, to be worthwhile, it is enough for an on-line algorithm to converge to a good static choice between W-I and W-U after a reasonable amount time. Their results are based on off-line algorithms while our competitive-update protocol is an on-line algorithm, so our study and theirs are complementing each other to cover a broad range of hybrid protocols.

The success of competitive-update schemes as shown in this paper comes from using a relaxed consistency model to hide the write latency. We have studied the detailed design issues involved in supporting multiple outstanding requests, in essence the design considerations for second-level lockup-free caches. Our study thus involves some of the issues studied by Gharachorloo *et al.* [17] to support relaxed consistency models but that paper does not investigate update-based protocols and its primary purpose is to compare the effectiveness of various memory consistency models. Their implementation models are referred to as BASIC, RDBYB, and LFC and correspond to our models RC-I, RC-I-bp, and RC-III-bp. While they show that bypassing of read misses in the first-level write buffer in conjunction with a lockup-free second-level cache is needed in order to fully hide the write latency in write-invalidate protocols, our study shows that read bypassing is actually not needed. This is an important contribution of this paper because read-bypassing complicates the write buffer design and can make it slower. As in their study, however, we confirm that only a single outstanding write request needs to be supported for write-invalidate protocols. However, as our study indicates, write-update protocols need lockup-free cache controller designs that can issue multiple outstanding write requests. Our evaluations of PTHOR, which uses fine-grain synchronization, show that Release Consistency exhibits better performance than Weak Ordering. This is in accordance to the results in [17]. We have extended their results by showing that this difference is even more pronounced for update-based protocols.

Update-based protocols have been considered in several distributed shared-memory systems where coherence is maintained at the page level [5, 34]. In [5], an architecture relying on software-controlled replication of pages and a write-update protocol in hardware for coherence maintenance is presented. The simulations show high processor efficiency over a range of applications running on up to 64 processors.

Wilson and LaRowe present a novel technique in [34] to maintain coherence of shared data at the page level. The technique is a hardware-supported but software-controlled mechanism that supports both invalidate and update-based protocols. The operating system software is responsible for choosing which coherence policy to use for each page. This study also shows that most write latency can be hidden by using a relaxed memory consistency model and by choosing an appropriate coherence policy for each page. Another study by the same authors [35] shows that as the block size increases write-update becomes preferable to write-invalidate in terms of memory traffic. Therefore, overall, write-update should be a much better choice than write-invalidate for page-level coherence in distributed shared memory systems.

In this study we show that competitive-update successfully reduces the read penalty for a wide range of applications as compared to write-invalidate. However, for applications with a high degree of migratory sharing competitive-update generates unnecessary write traffic, which may offset the read penalty reduction in networks with low bandwidths. Therefore, in [27], Nilsson and Stenström extend a competitive-update protocol with a previously published migratory detection mechanism [32]. The new adaptive protocol dynamically detects migratory data blocks and handles them with a read-exclusive policy. All other blocks are handled according to the competitive-update policy. They experimentally found that the adaptive protocol demands less than half of the network bandwidth as the competitive-update protocol for some of the applications with migratory objects (MP3D and Water). The reduction of network traffic is especially important because it makes the competitive-update policy suitable for an even broader range of multiprocessors.

A continuation of the work in this paper is presented in [11] where Dahlgren and Stenström propose to use a *write cache* as a means to reduce the write traffic associated with a competitive-update protocol. A write cache works in parallel with the second-level cache and is a small write-back cache with an allocate-on-write-miss and a no-allocate-on-read-miss strategy and a single valid/dirty-bit for each word. They evaluate the use of write caches together with a competitive-update protocol in a similar architectural model as in this study. They find a significant decrease in the write traffic and that a competitive threshold of one is sufficient when using a write cache together with a competitive-update protocol. Moreover, they find that most of the performance improvement is obtained with a very small (only four blocks) and direct-mapped write cache.

Finally, we speculate that the trends for larger systems are as follows. As more processors are added to a multiprocessor system, network latencies are expected to be longer. We believe that it is easier to achieve scalable bandwidth than scalable latencies, e.g., as a result of physical distances between processor nodes. This may impact the read and write penalties for the applications making it even more important to reduce the cache miss rate as much as possible. It is also likely that update-based protocols may require more extensive buffering as network latencies grow. We also believe that the communication demand and synchronization overhead will increase as the number of processors increases. It is interesting to note that a competitive-update protocol has a

potential to significantly reduce the read latencies as compared to a write-invalidate protocol, but at some increase in network traffic. As a result, we believe that competitive-update protocols will become even more favorable when the system size increases.

7 Conclusion

In this paper we analyze the relative performance of three different coherence policies: write-invalidate, write-update, and competitive-update. While previous studies have addressed the relative performance mainly in bus-based systems, we consider in this paper a cache-coherent NUMA architecture with a directory-based mechanism as a basis for the cache coherence protocols. Based on program-driven simulations of a detailed multiprocessor system and four benchmarks from the SPLASH suite we find that, contrary to what has been thought earlier, write-update cache-coherence protocols augmented with simple competitive mechanisms, referred to as competitive-update, have a potential to reduce the read penalty. We show that a competitive-update scheme can reduce the read penalty by as much as 46% as compared to write-invalidate. While it increases the traffic by 27% to 85%, this extra traffic did not offset the reduction in read penalty as a result of coherence-miss reduction. A negative effect appearing in one application, however, is the increase in synchronization overhead: Since release requests take a longer time to be globally performed as a result of a larger number of global writes, the acquire stall-time may increase in applications exhibiting contention for critical sections. We also found that update-based protocols, such as competitive-update, are more sensitive to the choice of consistency models than write-invalidate protocols. We found for two of the applications that competitive-update could perform as much as 9% better under Release Consistency than under Weak Ordering.

The two-level cache hierarchy organization we have adopted is compatible with high-performance microprocessors because it uses a simple, and thus fast, first-level cache and associates all protocol issues and lockup-free mechanisms with the second-level cache controller. Based on three buffering alternatives in the second level cache, we find that all performance benefits of Release Consistency can be exploited by allowing only a single outstanding write request in addition to a pending read-miss request under write-invalidate. However, in order to hide the write latency for the increased number of global write actions under update-based protocols, multiple outstanding requests are needed, although we did not see any use for more than 16 outstanding writes. We also studied the potential of letting read misses from the first-level cache bypass the first-level write buffer. We did not see any significant performance improvement from this design option. This is an important observation because it makes it possible to design a simpler and faster write buffer, which will scale with the processor speed.

This study suggests that update-based protocols augmented with a simple competitive mechanism can reduce the read-latency by reducing the number of misses and the latency of the remaining misses. However, since they trade the miss reduction for a larger number of global writes, they require relaxed consistency models to be effective. On the premise that the programming

community accepts the use of relaxed memory consistency models, we feel that the techniques presented in this paper are important to achieve the goal of scalable shared-memory systems but there is room for additional improvements.

References

[1] S.V. Adve and M.D. Hill, Weak ordering — A new definition, *Proc. 17th Internat. Symp. on Computer Architecture*, Seattle, WA (May 1990) 2-14.

[2] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz, APRIL: A processor architecture for multiprocessing, *Proc. 17th Internat. Symp. on Computer Architecture*, Seattle, WA (May 1990) 104-114.

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

[4] J.K. Archibald, A cache coherence approach for large multiprocessor systems, *Proc. Internat. Conf. on Supercomputing*, St. Malo, France (Jul. 1988) 337-345.

[5] R. Bisiani and M. Ravishankar, PLUS: A distributed shared-memory system, *Proc. 17th Internat. Symp. on Computer Architecture*, Seattle, WA (May 1990) 115-124.

[6] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, Portable programs for parallel processors (Holt, Rinehart and Winston, Inc., New York, NY, 1987).

[7] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, The Cachemire test bench — A flexible and effective approach for simulation of multiprocessors, *Proc. 26th Annual Simulation Symp.*, Arlington, VA (Mar. 1993) 41-49.

[8] D. Callahan, K. Kennedy, and A. Porterfield, Software prefetching, *Proc. Fourth Internat. Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, Santa Clara, CA (Apr. 1991) 40-51.

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

[10] F. Dahlgren, M. Dubois, and P. Stenström, Fixed and adaptive sequential prefetching in shared memory multiprocessors, *Proc. 1993 Internat. Conf. on Parallel Processing*, Vol I, Chicago, IL (Aug. 1993) 56-63.

[11] F. Dahlgren and P. Stenström, Reducing the write traffic for a hybrid cache protocol, *Proc. 1994 Internat. Conf. on Parallel Processing*, Chicago, IL (Aug. 1994). To appear.

[12] M. Dubois, C. Scheurich, and F. Briggs, Memory access buffering in multiprocessors, *Proc. 13th Internat. Symp. on Computer Architecture*, Tokyo, Japan (Jun. 1986) 434-442.

[13] M. Dubois and C. Scheurich, Memory access dependencies in shared memory multiprocessors, *IEEE Trans. Software Engrg.* SE-16(6) (Jun. 1990) 660-674.

[14] S.J. Eggers and R.H. Katz, A characterization of sharing in parallel programs and its application to coherency protocol evaluation, *Proc. 15th Internat. Symp. on Computer Architecture*, Honolulu, HA (May 1988) 373-382.

[15] S.J. Eggers and R.H. Katz, Evaluating the performance of four snooping cache coherency protocols, *Proc. 16th Internat. Symp. on Computer Architecture*, Jerusalem, Israel (May 1989) 2-15.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proc. 17th Internat. Symp. on Computer Architecture*, Seattle, WA (May 1990) 15-26.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, Performance evaluation of memory consistency models for shared-memory multiprocessors, *Proc. Fourth Internat. Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, Santa Clara, CA (Apr. 1991) 245-257.

[18] K. Gharachorloo, A. Gupta, and J. Hennessy, Hiding memory latency using dynamic scheduling in shared-memory multiprocessors, *Proc. 19th Internat. Symp. on Computer Architecture*, Gold Coast, Australia (May 1992) 22-33.

[19] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber, Comparative evaluation of latency reducing and tolerating techniques, *Proc. 18th Internat. Symp. on Computer Architecture*, Toronto, Canada (May 1991) 254-263.

[20] A. Gupta and W-D. Weber, Cache invalidation patterns in shared-memory multiprocessors, *IEEE Trans. Comput.* C-41(7) (Jul. 1992) 794-810.

[21] E. Hagersten, A. Landin, and S. Haridi, DDM — A cache-only memory architecture, *IEEE Comput.* 25(9) (Sep. 1992) 44-54.

[22] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator, Competitive snoopy caching, *Proc. 27th Annual Symp. on Foundations of Computer Science* (Oct. 1986) 244-254.

[23] D. Kroft, Lockup-free instruction fetch/prefetch cache organization, *Proc. 8th Internat. Symp. on Computer Architecture*, Minneapolis, MN (May 1981) 81-87.

[24] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Trans. Comput.* C-28(9) (Sep. 1979) 690-691.

[25] D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, The Stanford Dash multiprocessor, *IEEE Comput.* 25(3) (Mar. 1992) 63-79.

[26] T. Mowry and A. Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors, *J. Parallel Distributed Comput.* 12(2) (Jun. 1991) 87-106.

[27] H. Nilsson and P. Stenström, An adaptive update-based cache coherence protocol for reduction of miss rate and traffic, *Proc. Parallel Architectures and Languages Europe (PARLE) Conf.*, Athens, Greece (*Lecture Notes in Computer Science*, 817, Springer-Verlag, Berlin, Jul. 1994) 363-374.

[28] C. Scheurich, Access ordering and coherence in shared-memory multiprocessors, Ph.D. Thesis, University of Southern California, Los Angeles, CA, May 1989 (also U.S.C. Tech. Rep. CENG 89-19).

[29] J-P. Singh, W-D. Weber, and A. Gupta, SPLASH: Stanford parallel applications for shared-memory, *ACM SIGARCH Computer Architecture News* 20(1) (Mar. 1992) 5-44.

[30] P. Stenström, A survey of cache coherence schemes for multiprocessors, *IEEE Comput.* 23(6) (Jun. 1990) 12-24.

[31] P. Stenström, F. Dahlgren, and L. Lundberg, A lockup-free multiprocessor cache design, *Proc. 1991 Internat. Conf. on Parallel Processing*, Vol. I, Chicago, IL (Aug. 1991) 246-250.

[32] P. Stenström, M. Brorsson, and L. Sandberg, An adaptive cache coherence protocol optimized for migratory sharing, *Proc. 20th Internat. Symp. on Computer Architecture*, San Diego, CA (May 1993) 109-118.

[33] J.E. Veenstra and R.J. Fowler, A performance evaluation of optimal hybrid cache coherency protocols, *Proc. Fifth Internat. Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, Boston, MA (Oct. 1992) 149-160.

[34] A.W. Wilson, Jr., and R.P. LaRowe, Jr., Hiding shared memory reference latency on the Galactica Net distributed shared memory architecture, *J. Parallel Distributed Comput.* 15(4) (Aug. 1992) 351-367.

[35] A.W. Wilson, Jr., R.P. LaRowe, Jr., and M.J. Teller, Hardware assist for distributed shared memory, *Proc. 13th Conf. on Distributed Computing Systems*, Pittsburgh, PA (May 1993) 246-255.

[36] R. Zucker and J-L. Baer, A performance study of memory consistency models, *Proc. 19th Internat. Symp. on Computer Architecture*, Gold Coast, Australia (May 1992) 2-12.