To appear in Journal of Parallel and Distributed Computing, 1996

Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection

Håkan Grahn[1] and Per Stenström[2]

[1]Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
Hakan.Grahn@ide.hk-r.se, http://www.ide.hk-r.se/~nesse/

[2]Department of Computer Engineering, Chalmers University of Technology
S-412 96 Göteborg, Sweden
pers@ce.chalmers.se, http://www.ce.chalmers.se/~pers/

Abstract

Although directory-based write-invalidate cache coherence protocols have a potential to improve the performance of large-scale multiprocessors, coherence misses limit the processor utilization. Therefore, so called competitive-update protocols — hybrid protocols that on a per block basis dynamically switch between write-invalidate and write-update — have been considered as a means to reduce the coherence miss rate and have been shown to be a better coherence policy for a wide range of applications. Unfortunately such protocols may cause high traffic peaks for applications with extensive use of migratory objects. These traffic peaks can offset the performance gain of a reduced miss rate if the network bandwidth is not sufficient.

We propose in this study to extend a competitive-update protocol with a previously published adaptive mechanism that can dynamically detect migratory objects and reduce the coherence traffic they cause. Detailed architectural simulations based on five scientific and engineering applications show that this adaptive protocol outperforms a write-invalidate protocol by reducing the miss rate and bandwidth need by up to 71% and 26%, respectively.

Keywords: Cache coherence protocols, Memory consistency models, Performance evaluation, Shared-memory multiprocessors.

Proposed running head: A Competitive-Update Cache Protocol for Migratory Data

Corresponding author: Håkan Grahn, Phone: +46-457-787 62, Fax: +46-457-271 25

1. Introduction

Private caches in conjunction with a directory-based cache coherence protocol constitute an effective approach to reduce memory system latencies in large-scale shared-memory multiprocessors [17]. Such protocols maintain consistency by either invalidating [8], called *write-invalidate*, or updating [14], called *write-update*, remote copies when a local copy is modified.

Write-update protocols are not appropriate since they may incur a severe performance degradation as compared to write-invalidate protocols as a result of heavy network traffic. However, a previous study [9] has shown that a competitive-update protocol, a hybrid between write-invalidate and write-update protocols, outperforms write-invalidate protocols under relaxed memory consistency models [7] for a wide range of applications because of a lower miss rate. The idea of the competitive-update protocol is very simple. Instead of invalidating a copy of the block at the first write by another processor, the copy is updated. If the local processor does not access the copy it is invalidated after a number of global updates determined by a *competitive threshold*. As a result, only those copies regularly accessed are updated.

Although competitive-update protocols have better performance they can be suboptimal for coherence maintenance of migratory objects [10]. The reason is that migratory objects are often accessed in a read-modify-write manner by each processor in turn, i.e., a processor first reads the shared block and then updates the other copies of the block, then another processor reads and updates the block. Thus, the block will *migrate* between caches. For a competitive-update protocol, there is a risk that updates are sent to caches whose processors will not access the block until it has been invalidated due to the competitive threshold. This may cause unnecessary traffic that can increase the read penalty, i.e., the time the processors must stall due to cache misses, for networks with insufficient bandwidths. Therefore, write-invalidate is a better policy for migratory blocks.

To reduce the traffic of competitive-update protocols, and still maintain a low miss rate, we propose in this work to extend them with a previously published mechanism [4, 20] that dynamically detects memory blocks exhibiting migratory sharing. Such blocks are handled with read-exclusive requests to avoid unnecessary network traffic, while all other blocks are still handled according to the competitive-update policy.

Based on a detailed architectural simulation study using five parallel applications from the SPLASH benchmark suite [16] we find that competitive-update protocols extended with a simple migratory detection mechanism can reduce the miss rate by up to 71% as compared to a write-invalidate protocol. Our experimental results also show that the bandwidth requirements of the applications are reduced by up to 26% as compared to a write-invalidate protocol for applications exhibiting migratory sharing. The miss rate and traffic reduction help reducing the read penalty by up to 42% as compared to a write-invalidate protocol. As compared to a competitive-update protocol without the migratory detection mechanism, the bandwidth requirements are reduced by up to 62% for applications with migratory data. While parts of this work are presented in [15], we extend that study with a more detailed evaluation of the different protocol alternatives.

The rest of the paper is organized as follows. As a background, we begin in the next section by defining what migratory sharing is and how previously proposed cache coherence policies act with respect to migratory sharing. This serves as a motivation for the adaptive protocol we propose in Section 3. We move on to the experimental evaluation in Sections 4 and 5 starting with the experimental methodology, the detailed architectural assumptions, and the benchmark programs used in Section 4 and the experimental results in Section 5. Finally, we compare our findings with work by others in Section 6 and conclude the study in Section 7.

2. Cache Coherence Protocols and Migratory Sharing

In this section we first discuss migratory sharing and the access pattern caused by this type of sharing behavior. Then we describe a write-invalidate protocol and discuss the implications migratory objects have on the performance of write-invalidate protocols. Finally, we describe how the competitive-update protocol works and its performance limitations with respect to migratory objects.

Our architectural framework consists of a cache-coherent NUMA (Non-Uniform Memory Access) architecture [19] with a directory-based cache coherence protocol which is described in detail in Section 3. In such an architecture the processor stall times due to completion of memory accesses limit the performance of the whole system. The stall times stem mainly from two reasons; the read stall time arises mainly from cache misses while the write stall time arises when propagating new values to remote copies upon processor writes. Previous studies have shown that it is possible to hide all write latency, and thus remove all write stall time, by using a relaxed memory consistency model, e.g., Release Consistency (RC) [6], and sufficient amount of buffering in the local node. This has been shown possible both under write-invalidate protocols [7] and under competitive-update protocols [9]. In order to achieve high performance we assume in this study Release Consistency to be able to hide all write latency.

2.1. Migratory Sharing

Gupta and Weber classify data objects based on the access pattern they exhibit in [10]. *Migratory data objects* are such data structures that are manipulated by only a single processor at any given time. When such a data object is manipulated by another processor it is said to migrate from one processor to another. In parallel programs such objects are not unusual, e.g., data structures that are accessed in critical sections and high-level language statements such as `I:=I+1` exhibit migratory sharing.

A way of formally defining migratory sharing is as a sequence of read-modify-write actions by alternating processors on a data object. The global reference stream to a migratory object can then be described by the following regular expression:

$$\ldots (R_i)\,(R_i)^*\,(W_i)\,(R_i \mid W_i)^*\,(R_j)\,(R_j)^*\,(W_j)\,(R_j \mid W_j)^* \ldots \quad (1)$$

In the expression above R_i and W_i represent a read access and a write access, respectively, by processor *i*, '*' denotes zero or more occurrences of the preceding string, and '|' denotes the logical OR-operation. As we will see, the difference in which the protocols respond to migratory sharing stems from the fact that there is at least one R_i followed by at least one W_i by the same processor, *i*, before the next processor, *j*, starts accessing the block in the same way.

2.2. Write-Invalidate Protocols and Migratory Sharing

Write-invalidate protocols rely on invalidation messages to maintain coherence among cached copies of a shared memory block. We describe the actions associated with a write-invalidate protocol and refer to the regular expression (1). We assume that the block is valid in memory before the read request R_i is issued. R_i results in a read-miss request since the block is not present in the cache. The block is loaded into cache *i* as shared. Upon the write request W_i to the shared block, the copy in memory is invalidated and processor *i* receives an exclusive (dirty) copy of the block. Subsequent writes by processor *i* can be performed locally until another processor accesses the block.

When processor *j* reads the block (R_j) a cache miss is encountered and a read-miss request is forwarded to cache *i* which has the exclusive copy. Cache *i* then sends a copy to cache *j* and the block becomes shared again. Later, when processor *j* modifies the block (W_j) the modification results in a *single* invalidation message sent to cache *i*, i.e., the cache that had the block exclusive before. An optimization is to merge the read-miss request and invalidation request into a single read-exclusive request. Two previous papers have studied this *migratory detection mechanism* [4, 20]. In both papers migratory blocks are dynamically detected by the hardware and are handled by read-exclusive requests instead of separate read-miss and invalidation requests. This optimization reduces the network traffic because all single invalidations to migratory blocks are removed. However, the coherence miss rate is unaffected.

In [20], the merits of the migratory detection mechanism were evaluated in a similar architectural framework that we use in this study. They found that the write-invalidate protocol extended with the migratory detection mechanism reduces the number of global write requests by up to 96% for applications exhibiting migratory sharing. Under a relaxed memory consistency model, this optimization results in that the read penalty can be reduced as a result of lower contention, especially for applications with higher network bandwidth requirements than the network can sustain. In fact, the network traffic was found to be reduced by more than 20% for the studied applications that exhibit migratory sharing.

2.3. Competitive-Update Protocols and Migratory Sharing

In competitive-update protocols coherence is maintained by update messages rather than invalidation messages. Upon a write request, update messages are sent to all caches sharing the same memory block. In contrast to write-update protocols, a *competitive threshold*, C, is used to locally invalidate copies that are not accessed by the local processor between a number of updates, i.e., when a copy has been updated C times it is invalidated and the update messages to it cease. Thus, the network traffic is reduced as compared to a write-update protocol since only those copies regularly accessed are updated. Although more details on how competitive-update protocols work and a detailed performance evaluation are found in [9], we discuss the main results below.

The performance evaluation in [9] shows that competitive-update protocols can reduce the read penalty by up to 46% as compared to write-invalidate protocols, mainly as a result of a highly reduced coherence miss rate. The coherence miss rate is reduced by up to 76% as compared to a write-invalidate protocol. A drawback of competitive-update protocols is an increased number of global writes. However, under a relaxed memory consistency model this extra write traffic was shown not to offset the reduced read penalty given a sufficient network bandwidth.

Unfortunately, for memory blocks that migrate between several caches, competitive-update protocols work suboptimally since cached copies are often updated a large number of times (without any local access), and thus they are invalidated. This results in unnecessary network traffic. For networks with insufficient bandwidth, contention may offset the benefits of the competitive-update protocol. To address this problem we propose in this study to extend the competitive-update protocol with the previously published migratory detection mechanism [20] discussed in Section 2.2. One would expect that such an extended protocol reduces the traffic caused by blocks exhibiting migratory sharing, while still maintaining the same low coherence miss rate as in the competitive-update protocol for blocks that do not exhibit migratory sharing. In the next section we describe the details regarding the migratory detection mechanism and how it is incorporated in the competitive-update protocol.

3. The Proposed Adaptive Protocol

We start in Section 3.1 to present the architecture we use as a base for the implementation and performance evaluation of our protocol. Then in Section 3.2 we provide a detailed description of the competitive-update protocol and in Section 3.3 we describe how to extend the competitive-update protocol with the migratory detection mechanism.

3.1. The Simulated Multiprocessor Architecture

As a base for our protocol evaluation, we assume a cache coherent non-uniform memory access (CC-NUMA) architecture. It consists of a number of processor nodes interconnected by a general network. The organization of a processor node and an overview of the simulated system architecture are shown in Fig. 1.

A processor node consists of a processor with a two-level cache hierarchy and associated write buffers. The cache hierarchy is interfaced to the local portion of the shared memory and the Network Interface Control (NIC) by a local bus according to Fig. 1. Global cache coherence is supported by a directory-based protocol according to Censier and Feautrier [3]; each memory block associates a presence-flag vector indicating which nodes have a copy of the block.

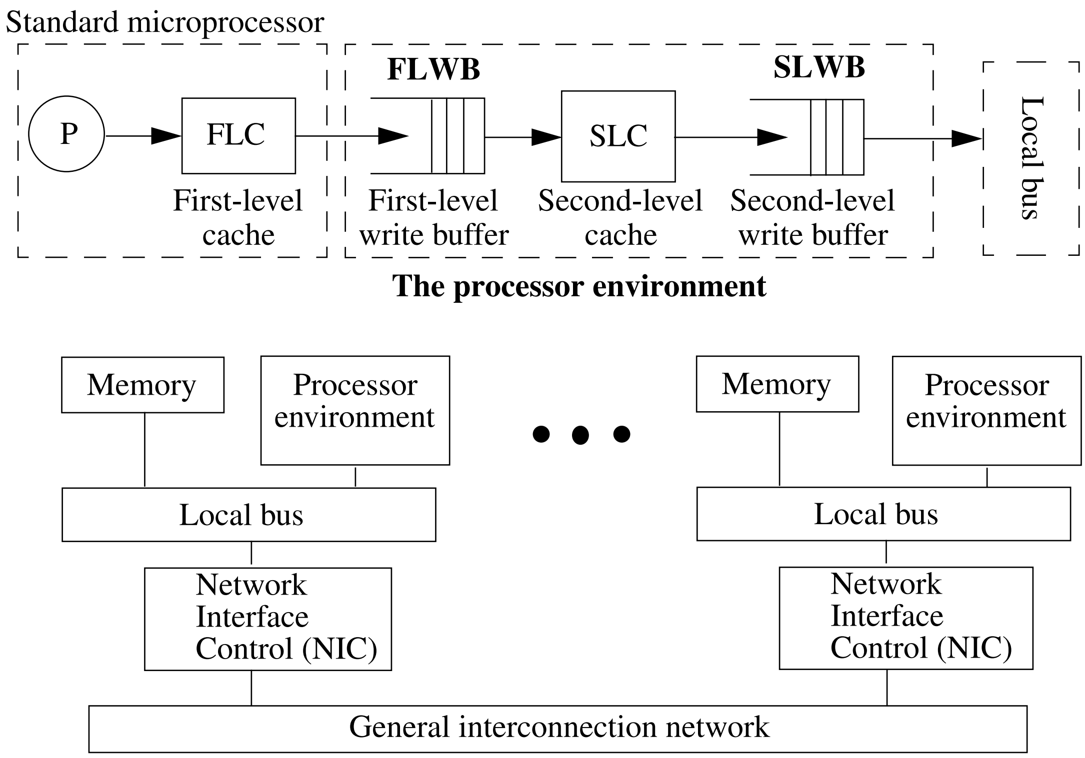

Fig. 1. The processor environment and the simulated architecture.

The first-level cache (FLC) is a write-through on-chip cache whereas the second-level cache (SLC) is a copy-back cache. Both caches are direct-mapped with the same line size in both caches and full inclusion is supported; if a block is present in the FLC it is also present in the SLC. The SLC is lockup-free [12, 18] whereas the FLC is blocking and has an invalidation pin so a block can be invalidated from outside the processor. All coherence actions associated with the system-level cache coherence protocol are handled by the SLC and by the memory controller.

3.2. The Competitive-Update Protocol

In the competitive-update protocol, each memory block can be in two different stable states: *Present* (a clean copy resides in the memory module) and *Modified* (exactly one cache has an exclusive copy of the block). In addition, transient states dictate, e.g., if updates are pending as a result of a global write request. Each cache block can be in one of three states: *Invalid* (the cache does not have a copy of the block), *Shared* (other copies of the block exist in the system), and *Exclusive* (the cache has the only copy of the block in the system). A counter, indicating how many external updates have been received since the last access by the local processor to the block, is associated with each SLC cache block. We now describe how read and write accesses are handled by the protocol. We will refer to the node where the page containing the block is allocated as *home*, *local* as the node from which the request originated, and finally *remote* as any other node involved in the coherence action.

Processor read accesses, that miss in the FLC, to a cache block in states Shared or Exclusive are satisfied by the SLC and the counter is preset to the competitive threshold. If the block is Invalid, as in Fig. 2, an SLC read miss occurs and a global read request (GRd) is sent to home. If home is the local node and if the memory block is clean, the miss is serviced locally in the processor node. Otherwise, the miss is serviced either in two (to the left in Fig. 2) or four (to the right in Fig. 2) node-to-node traversals depending on whether the memory block is in state Present or Modified. If the block is in state Modified, memory must be updated by an UMem message before the read request is serviced. When the requesting cache receives the block through the Data message, it is loaded into the cache in state Shared and the counter is preset to the competitive threshold. The memory block ends up in state Present after the global read request is satisfied.

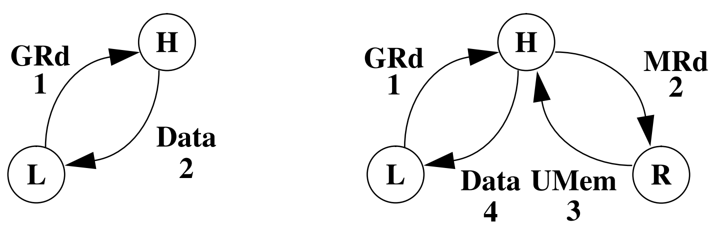

Fig. 2. A read miss is serviced in two (left) or four (right) node-to-node traversals depending on whether the block is in state Present or Modified in memory.

A processor write access is serviced locally in the SLC if the cache block is in state Exclusive and the counter is preset to the competitive threshold. When the cache block is in state Shared or Invalid, the other copies of the block must be notified of the write. According to Fig. 3, a global write request along with the data (GWr) is issued from the local node to home. The home memory controller is responsible for sending explicit update messages (CUp) to each node with a copy according to the state of the presence-flag vector and the global coherence state of the block. Upon the reception of a CUp message, the SLC controller checks the counter associated with the block. If the counter is zero the block is invalidated in both the FLC and the SLC, and a CIAck is sent to home. Otherwise, the counter is decremented, the SLC copy is updated and the FLC copy is invalidated, and a CAck is sent to home indicating that the cache block is still to be updated. When home has received all acknowledgments, it decides whether local shall receive an exclusive copy (WrAckE) or not (WrAck), based on the existence of any copies left in other caches. When the write acknowledgment is received by local, the counter is initialized to the competitive threshold. The memory block ends up in state Modified or Present depending on whether there is an exclusive copy in the system or not.

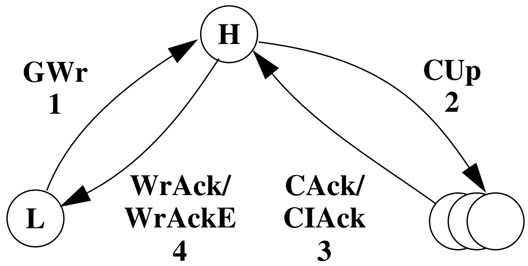

Fig. 3. Coherence actions taken upon a write request.

The competitive-update protocol mimics the behavior of a write-invalidate protocol when the competitive threshold is zero, i.e., the CUp message always causes the cache copy to be invalidated. The only difference is that the GWr and CUp messages contain data (the modified word of a block) in the competitive-update protocol whereas in a write-invalidate protocol this would not be needed. In our experimental section we use a write-invalidate protocol as the basis for comparison. The write-invalidate protocol used in our experiments in Section 5 has the same coherence actions as a competitive-update protocol with a competitive threshold of zero, but data is not contained in the GWr and CUp messages.

3.3. Extending the Competitive-Update Protocol with the Migratory Detection Mechanism

In order to identify migratory blocks, our migratory detection mechanism — which is conceptually the same as in [20] — detects the sequence $W_i\ R_j\ W_j$ and classifies a memory block as migratory when W_j occurs. In this section we identify the hardware mechanisms and the protocol extensions that incorporate this detection mechanism in the competitive-update protocol, starting with a high-level view of the previously published migratory detection mechanism [20].

3.3.1. A High-Level View of the Migratory Detection Mechanism

In [20], Stenström *et al.* distinguish between migratory blocks and ordinary blocks, i.e., those blocks that do not exhibit migratory sharing. In their study, cache coherence for ordinary blocks is maintained by a write-invalidate protocol.

For migratory blocks, home converts a read-miss request to a read-exclusive request. This request is then forwarded to remote, i.e., the cache with an exclusive copy of the block. Remote sends a copy to local and invalidates its own copy of the block. Local then loads the exclusive copy of the block into its cache. As a result, subsequent processor writes by the requesting processor can be performed locally and all explicit invalidations are eliminated.

If all blocks are classified as ordinary by default, home must detect when a block starts to be migratory, i.e., when home sees the following request sequence: $R_i\ W_i\ R_j\ W_j$. By letting home keep track of the processor that most recently modified the block, i.e., i, the block is classified as migratory when home receives a global write from processor j (W_j) given that the following two requirements are fulfilled:

Condition 1 (Migratory detection for write-invalidate protocols)

1. $j \neq i$; the processor that issues the write request is not the same as the processor that most recently issued a write request to the block.
2. The number of block copies is exactly two.

The hardware requirement is a last-writer pointer (LW) of size $\log_2 N$ bits, given N caches in the system, to keep track of the identity of the processor that last modified the block and a bit denoting whether the memory block is migratory or not. In addition, for full-map protocols the number of copies can be derived from the content of the presence-flag vector.

The detection mechanism described above is not directly applicable to a competitive-update protocol because in a competitive-update protocol it is not sufficient to know the number of block copies; a copy of the block may exist in a cache, although the local processor has not accessed it since the last global modification by another processor. We will therefore reformulate Condition 1 in the next section to serve as a basis for how the detection mechanism is adjusted to fit the competitive-update protocol.

3.3.2. Adopting the Migratory Detection Mechanism in the Competitive-Update Protocol

Given sequence (1) from Section 2.1., the migratory detection mechanism classifies the sequence as migratory when processor j issues the write request (W_j). At this point, there may exist an arbitrary number of copies, but at least two. Therefore, as a base for the detection algorithm, we reformulate requirement 2 in Condition 1:

Condition 2 (Migratory detection for competitive-update protocols)

1. $j \neq i$; the processor that issues the write request is not the same as the processor that most recently issued a write request to the block.
2. Processors i and j are the only ones that may have read from the block since the write from processor i.

Requirement 1 is the same for competitive-update protocols as for write-invalidate protocols. Therefore, we associate a *last-writer* pointer (LW) with each memory block also in this case. The LW pointer is updated on each global write request. As for requirement 2, however, the algorithm detects whether processor i and processor j are the only ones that have read the block since the write from processor i by letting home ask all caches with a copy whether their processors have read the block since they detected the write from processor i. As we show in Fig. 4, this question round does *not* generate more network traffic than the write-invalidate protocol. By checking the counter associated with the block, each cache locally decides whether or not the processor has read the block after the last write by another processor. However, since the counter is also preset when the local processor writes to the block, an additional state is needed to determine whether the last update came from another processor or not. Home keeps track of migratory and ordinary memory blocks by a new stable state for memory blocks and a new transient state.

When a processor, *i*, writes to a block that no other processor has updated since the last read by *i*, the block is deemed as a potentially migratory block. As we see to the left in Fig. 4, a MigrWr message is sent to home instead of an ordinary global write. Home checks whether (LW = *i*). If (LW = *i*) the write is treated as an ordinary write. Otherwise, the memory block in home agrees that the block is potentially migratory and sends out MigrInv messages to those caches sharing the block. A cache *k* responds to MigrInv in two ways; (i) *k* agrees that the block is migratory (MOk) if processor *k* has not read the block since the last update or if the last global update originated from *k*, or (ii) *k* disagrees (MNotOk) if the processor has read but not written to the block since the last update by another processor. Home classifies the block as migratory iff all acknowledgments are MOk. Then, exclusive ownership is given to local by MWrAck. By contrast, if at least one cache responds with MNotOk, the block is still classified as ordinary and a WrAck is sent to local. Note that no extra network traffic, as compared to the write-invalidate protocol, is needed in order to detect migratory blocks.

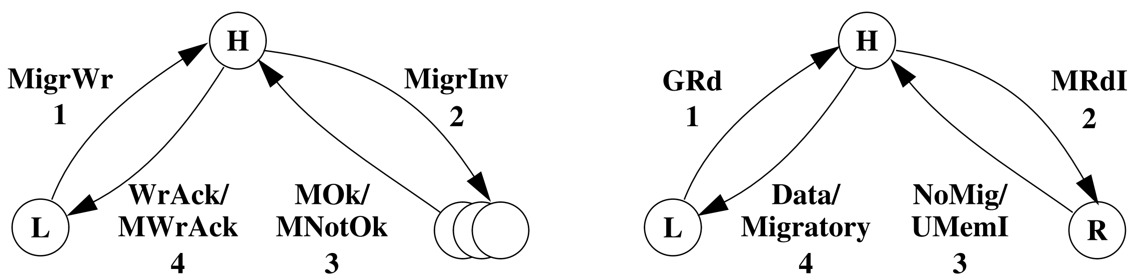

Fig. 4. Coherence actions for detection of migratory blocks (left) and coherence actions for read misses to migratory blocks (right).

Read-miss requests to migratory blocks are handled according to the right part of Fig. 4. Local issues a GRd, as for ordinary blocks. Home knows that the block is migratory and requests (MRdI) remote to send its exclusive copy back to home and invalidate its own copy. Remote responds with UMemI to home, which forwards an exclusive copy to local by the Migratory message.

Like the protocol in [20], a new cache state called *Migrating* is needed to detect when a block discontinues to be migratory. For migratory blocks, home only sees read-miss requests and no write requests, i.e., home only sees the reference sequence R_i R_j R_k. As a result, home can not detect whether a processor has modified the block between two subsequent read-miss requests. To solve this problem, a migratory block is loaded into the cache in state Migrating upon a read-miss. When the local processor modifies the block the state changes to Exclusive without any global actions. Later, when remote receives MRdI from home, it decides whether the block is still migratory or not. If the block is in state Exclusive, it remains migratory and UMemI is sent to home. By contrast, if the block is in state Migrating, i.e., the processor has not modified it, when MRdI arrives, the block is no longer migratory. Remote sends NoMig to home and keeps a shared copy of the block. Home reclassifies the block as ordinary and Data is sent to local. Finally, the block is loaded into the cache as shared.

In summary, the extra hardware needed to detect migratory blocks beyond what is already there to support the competitive-update protocol is one pointer associated with each memory block and one extra bit associated with each cache block. The competitive-update protocol is extended with two memory states and one local cache state.

3.3.3. An Enhanced Adaptive Protocol

In applications with false sharing [5] where two processors are involved, a problem can arise in the adaptive protocol discussed in Section 3.3.2. Consider the following reference sequence to a block, where a block is alternatively accessed by processor i and j:

$$\ldots (R_i)\ (W_i)\ (R_j)\ (R_i)\ (W_j)\ (R_i)\ (R_j)\ (W_i)\ldots \quad (2)$$

The above sequence is detected as migratory at the time processor j issues W_j. However, the subsequent read access by processor i results in a read-exclusive copy of the block which then leads to a cache miss by processor j (R_j) that would have been avoided under a write-invalidate protocol. Thus an increased number of misses can occur. We have observed this anomaly of the migratory detection mechanism for one of the applications we experimentally study (Ocean).

Our approach to mitigate the problem is to resort to competitive-update mode when exactly two processors share a block. To do this a *last-last-writer* pointer, referred to as LLW, is associated with each memory block in addition to the previous LW pointer. When the LW pointer is updated with a new value, the old value of LW is moved to the LLW pointer. In order to classify a block as migratory, the processor that currently modifies the block, say i, must not be any of the last two ones that modified the block, i.e., ($i \neq$ LW) and ($i \neq$ LLW).

4. Simulation Methodology, Architectural Parameters, and Benchmarks

In this section, we present the simulation framework in which we have studied the effectiveness of the adaptive protocol according to Section 3.3. In Section 4.1. we present the simulation environment and the detailed architectural assumptions, and in Section 4.2. we describe the benchmark programs.

4.1. Simulation Environment and Architectural Parameters

The simulation models are built on top of the CacheMire Test Bench [2]; a program-driven simulator and a programming environment. The simulator consists of two parts: a functional simulator of multiple SPARC processors and an architectural simulator. The functional simulator issues memory references, and the architectural simulator delays the processors according to its timing model. Thus, the same interleaving of memory references is obtained as in the target systems we model.

The organization of the architecture is shown in Fig. 1. The two-level cache hierarchy we simulate consists of a 2 Kbyte first-level cache (FLC) and an infinite second-level cache (SLC), both with a cache-line size of 16 bytes. The write buffers contain 16 entries each by default. Acquire and release requests are supported by a queue-based lock mechanism similar to the one implemented in the DASH multiprocessor [13]. The page size is 4 Kbyte and the pages are allocated to memory modules in a round-robin fashion; pages with consecutive page numbers are allocated to nodes with consecutive node identities.

As for the timing model, we consider SPARC processors and their FLCs clocked at 100 MHz (1 pclock = 10 ns). The SLC is assumed to be implemented by static RAM with an access time of 30 ns. The SLC and its write buffer are connected to the network interface control (NIC) and the local memory module by a 128-bit wide split transaction bus clocked at 33 MHz. Thus, it takes 30 ns to arbitrate for the bus and 30 ns to transfer a request or a block. Furthermore, the memory is assumed to be implemented by dynamic RAM with an access time of 90 ns including buffering.

We simulate a system containing 16 nodes interconnected by a 4-by-4 wormhole routed synchronous mesh with a flit size of 64 bits. To especially look at how the adaptive protocol manages to reduce network contention, we assume a fairly conservative mesh implementation that is clocked at 33 MHz by default. We correctly model contention for all parts in the system. Table I shows the time it takes to satisfy 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, however, requests will normally take a longer time as a result of contention.)

Table I
Latency numbers for processor read requests when data is supplied from different levels in the memory hierarchy.

Description	Latency (1 pclock = 10 ns)
Fill from FLC	1 pclock
Fill from SLC	4 pclocks
Fill from Local Memory	28 pclocks
Fill from Home (2-hop)	100 pclocks
Fill from Remote (4-hop)	196 pclocks

4.2. Benchmark Programs

In order to understand the relative performance of the adaptive protocol, we use five scientific and engineering applications, all taken from the SPLASH suite [16] except for Ocean that has been provided to us from Stanford University. The main characteristics of the five benchmark programs together with the size of the data set used are summarized in Table II. All programs are written in

C using the PARMACS macros from Argonne National Laboratory [1] and compiled with `gcc` version 2.1 with optimization level -O2. Previous studies [10, 20] have shown that MP3D, Cholesky, and Water have a high degree of migratory objects, while PTHOR and Ocean have many producer-consumer objects.

Table II
Benchmark programs

Benchmark	Description	Data Sets/Input
MP3D	Particle-based wind-tunnel simulator	10 000 particles, 10 time steps
Cholesky	Cholesky factorization of a sparse matrix	bcsstk14 matrix
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	Simulate eddy currents in an ocean basin	128-by-128 grid, tolerance 10^{-7}

5. Experimental Results

In this section we present our experimental results starting in Section 5.1. by pointing out the performance limitations of competitive-update protocols for applications with a significant amount of migratory sharing. In Section 5.2. we present the performance of the enhanced adaptive protocol according to Section 3.3.3., and finally in Section 5.3., we evaluate the relative merits of the basic and enhanced adaptive protocols according to Sections 3.3.2 and 3.3.3, respectively.

5.1. Performance Limitations of Competitive-Update Protocols

From a previous study [9] it is known that a competitive-update protocol reduces both the read penalty and the execution time as compared to a write-invalidate protocol for a wide range of applications given a sufficient network bandwidth. In that study, a 100 MHz mesh was assumed and we will first study the relative performance of competitive-update and write-invalidate protocols assuming the same experimental set-up.

In Fig. 5 the normalized execution times of the write-invalidate (WI) and the competitive-update (CU) protocols are presented for the five applications under study. Two bars are associated with each application and correspond to WI and CU, respectively. Each vertical bar is divided into four sections that correspond to (from the bottom to the 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. Note that there is no write stall time since we use Release Consistency which has shown to be able to hide all write latency [7, 9]. In our measurements, we have assumed a competitive threshold of 4. WI is implemented as described in the last paragraph of Section 3.2.

In Fig. 5 we see that CU reduces the execution time for all five applications by between 1% and 13% as compared to WI. The reason for the reduced execution time is that the read penalty is

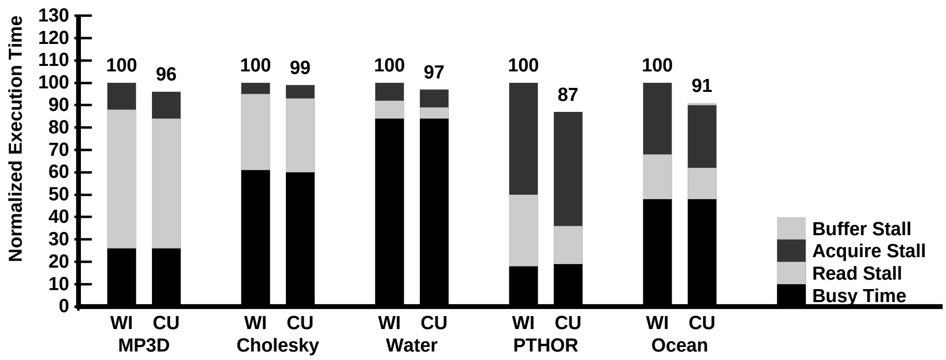

Fig. 5. Normalized execution times for the applications running on an architecture with a 100 MHz mesh.

cut by up to 46% (PTHOR) as a result of fewer coherence misses. Because of the large amount of migratory sharing, MP3D and Cholesky show the lowest read penalty reduction.

If a network with less bandwidth is considered, we expect the benefits of CU to be reduced as compared to WI for applications with a significant amount of migratory sharing as a result of the unnecessary write traffic. To study this, we henceforth assume the default mesh implementation that is clocked at 33 MHz. The normalized execution times for the applications in this case are found in Fig. 6. We observe that both MP3D and Cholesky now have longer read stall times under CU than under WI. The read penalty reduction under CU that we observed in Fig. 5 is now reversed to an increase in the read penalty.

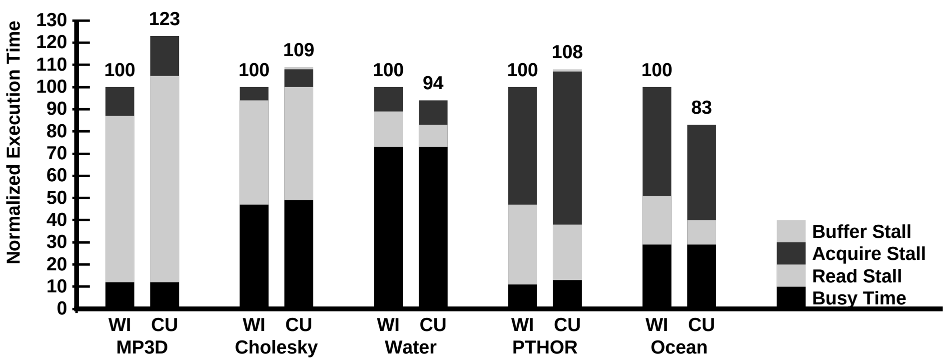

Fig. 6. Normalized execution times for the applications running on an architecture with a 33 MHz mesh.

In Fig. 7, the bandwidth requirement for each application is shown. The bandwidth requirement is the network traffic generated by an application divided by the execution time of the application. The whole bar corresponds to the bandwidth requirement of CU whereas the black section corresponds to the bandwidth requirement of WI; WI requires less bandwidth than CU for all applications. As expected, MP3D and Cholesky have more than twice as high bandwidth requirements as the other applications under CU. The large number of unnecessary updates of migratory objects cause this dramatic difference in bandwidth requirements. In addition, for MP3D we see that CU requires 51% more bandwidth than WI, and even worse for Cholesky where CU requires 86% more bandwidth than WI.

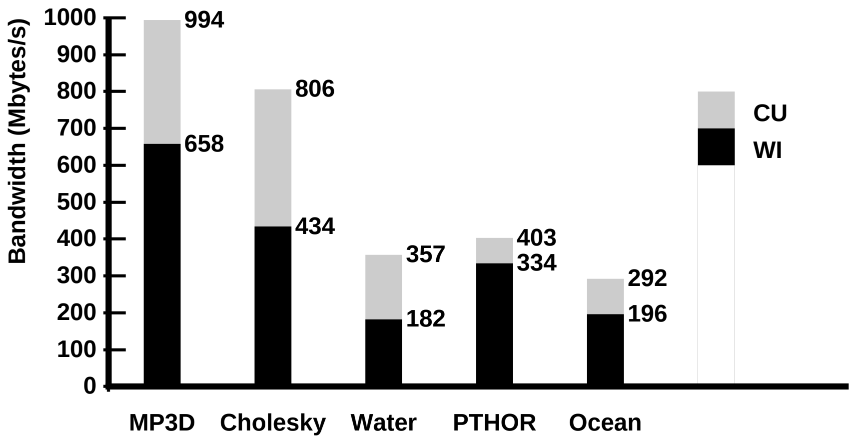

Fig. 7. Bandwidth requirements under WI and CU for each application running on an architecture with a 33 MHz mesh.

In Fig. 6, we also observe that the acquire stall time has increased significantly for PTHOR. This effect arises because PTHOR exhibits contention for critical sections, i.e., at the time a lock is released there is already another processor waiting to get the lock. The higher rate of global write actions under CU than under WI delays the issue of a release, which as a secondary effect delays the processors waiting for the lock. The adaptive protocol we propose can reduce both these problems as we will show in the next section.

5.2. Relative Performance of the Enhanced Adaptive Protocol

In this section we will analyze the performance of the enhanced adaptive protocol, henceforth referred to as AD+, described in Section 3.3.3. relative to WI and CU. In Fig. 8 the execution times of CU and AD+ are shown. The execution times are normalized to the execution time of WI (=100) for each application.

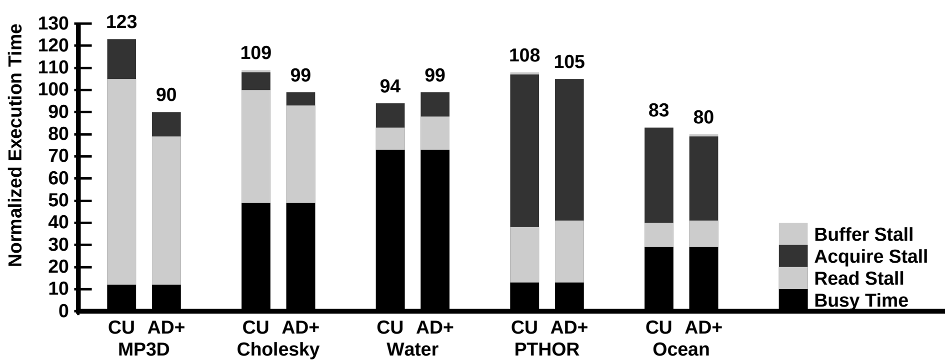

Fig. 8. Execution times for CU and AD+ normalized to the execution time for WI.

Overall, despite the low-bandwidth network, AD+ manages to perform better than WI for all applications but one (PTHOR). The execution times have decreased under AD+ by between 1% and 20% as compared to WI. Simulations show that for MP3D and Cholesky the main reason is that the read penalty has been reduced by 11% and 4%, respectively, as a result of lower contention in the network (not shown in the diagrams). Our results also show that the read penalty is reduced by 22% for PTHOR under AD+ as compared to WI. However, the acquire stall time under AD+ is 21% longer than under WI. The larger number of global write actions for ordinary blocks impact the time to issue a release adversely, and since PTHOR exhibits contention for critical sections this impacts the acquire stall time. This observation is consistent with the results in [9]. The highest read penalty reduction is achieved for Ocean where AD+ cuts 42% of the read penalty as compared to WI. For Ocean, AD+ also reduces the acquire stall time by 22% as compared to CU. In Ocean, the counters in the barriers exhibit migratory sharing. The fewer global write actions under AD+ result in a shorter release issuance time. As a result, the acquire stall time is reduced if another processor waits for the lock. Remarkably, although Water exhibits substantial migratory sharing, there is virtually no difference in performance between WI and AD+. The reason is that the bandwidth requirement for Water is very low (see Fig. 7) which means that the read penalty is not increased due to network contention under WI.

In Fig. 9, we show the miss rates for each application. Two bars are associated with each application: one for CU and one for AD+. Each bar is divided into two parts: the number of cold misses (the lower part) and the number of coherence misses (the upper part). The miss rates are normalized to the total miss rate under WI (=100) for each application. As expected, for MP3D and Cholesky the miss rates of AD+ are almost the same as under WI. Surprisingly, the miss rate for Water is 14% lower under AD+ than under WI even though most data objects are migratory. In AD+ three different processors must modify a memory block, one after the other with no intervening access by any other processor, in order to classify the block as migratory. If only one other processor modifies the block before the same processor accesses the block again the block will resort to competitive-update. Thus a lower coherence miss rate is encountered even though the block is migratory. For applications with marginal migratory sharing (PTHOR and Ocean) the miss rates are reduced under AD+ by 28% and 71%, respectively, as compared to WI. Hence, AD+ has preserved the low miss rates of CU for applications with little or no migratory sharing.

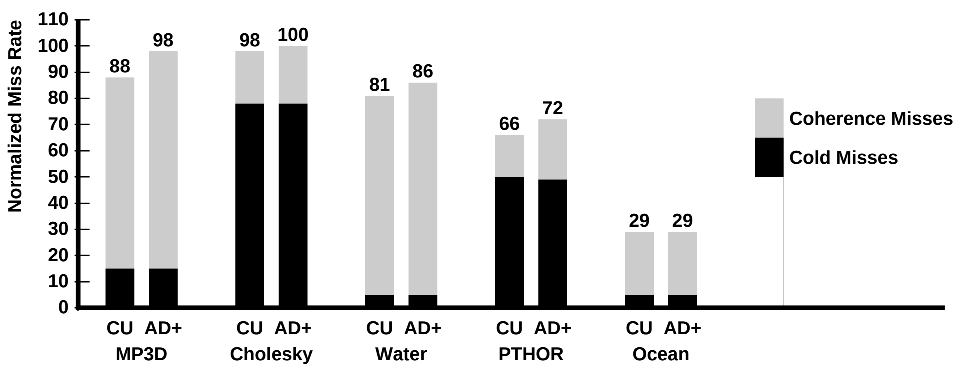

Fig. 9. Normalized miss rates for CU and AD+ as compared to WI.

As for the traffic reduction by AD+, we see in Fig. 10 that the bandwidth requirements for AD+ is significantly reduced as compared to WI, and even more as compared to CU, for MP3D, Cholesky, and Water. For these applications the bandwidth requirements are reduced by between 10% and 26% as compared to WI. Comparing AD+ and CU we find that the bandwidth requirements are reduced by between 51% and 62%. As expected, the bandwidth requirements are virtually the same under AD+ and CU for applications with little or no migratory sharing (PTHOR and Ocean).

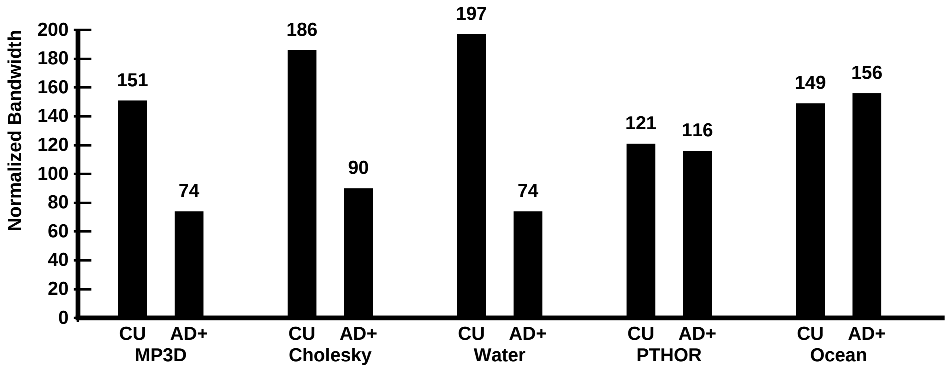

Fig. 10. Normalized bandwidth requirements for CU and AD+ as compared to WI.

To summarize, AD+ appears to be a better default policy than WI and CU because like WI it handles migratory blocks according to the read-exclusive strategy provided by the migratory detection mechanism which helps reducing traffic. Like competitive-update protocols, AD+ handles producer-consumer sharing (such as in Ocean and PTHOR) in competitive-update mode and helps reducing the coherence miss rate for such blocks. Finally, we have also varied the competitive threshold in AD+ and did not find any significant performance variations.

5.3. Comparing the Basic and the Enhanced Adaptive Protocol

In Section 3.3.2., we described an adaptive protocol, henceforth referred to as AD, which uses a single last-writer pointer per memory block. In AD, a LW pointer is used to keep track of the processor that most recently modified a memory block. The enhanced adaptive protocol we described in Section 3.3.3., referred to as AD+, is AD extended with a second pointer, a LLW pointer, to keep track of the last two processors that modified a memory block. If exactly two processors alternately modify a block, the block is classified as migratory in AD but not in AD+. We use Ocean as a case study because it exhibits a non-negligible degree of false sharing. Although false sharing should be avoided, it has shown to be a good trade-off in this application because the SOR algorithm used in this application converges faster.

In Fig. 11, we present the normalized execution times (the diagram to the left) and the miss rates (the diagram to the right) for Ocean under four coherence protocols: WI, CU, AD+, and AD. In the diagram over the miss rates in Fig. 11 there is a section called *classification misses*. Classification misses constitute a new type of cache misses that arise as a result of memory blocks that are erroneously classified as migratory. A block is detected as migratory in AD both for the reference sequence (1) in Section 2.1. *and* sequence (2) in Section 3.3.3. As a result, blocks that suffer from false sharing between two processors can be classified as migratory in AD even though it is preferable to resort to competitive-update mode. From Fig. 11, we see that the miss rate for AD is more than twice the miss rate for AD+, which clearly shows the usefulness of the LLW pointer.

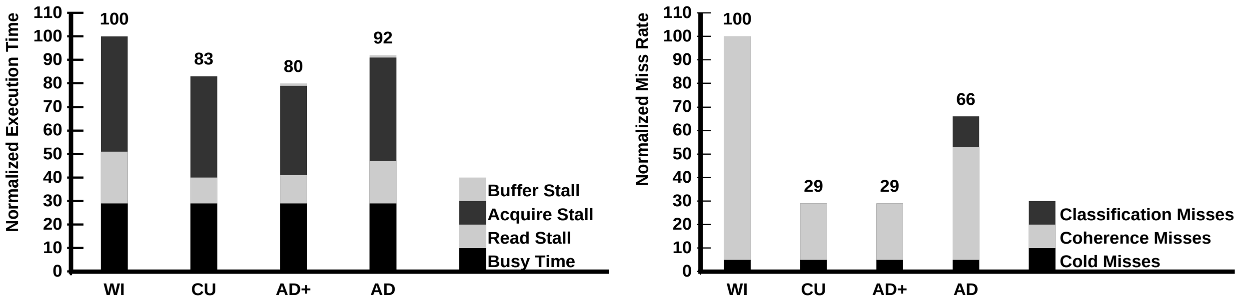

Fig. 11. Normalized execution times (left) and miss rates (right) for Ocean under four various protocols.

As the diagram to the left in Fig. 11 shows, the high miss rate for AD as compared to AD+ increases the execution time for AD as compared to AD+. The read penalty for AD has increased by 38% as compared to AD+. The acquire stall time under AD has also increased by 16% as a result of the higher network traffic due to the read misses. In total, the execution time is 15% longer under AD than under AD+ which removes most of the advantages of competitive-update protocols. We did not see any significant difference in performance between AD and AD+ for any of the other applications because they exhibit very little false sharing.

6. Discussion and Related Work

We have shown that for competitive-update protocols it is possible to reduce the bandwidth requirements by up to 26% as compared to a write-invalidate protocol by detecting migratory data objects. Although the architecture assumed in this study is a CC-NUMA with a mesh network, the detection mechanism is applicable also to bus-based systems where a low traffic rate is more important than in CC-NUMA architectures with mesh networks.

In [4], Cox and Fowler studied the migratory detection mechanisms needed both in directory-based protocols and in snoopy cache protocols. The migratory detection mechanism they propose for the directory-based protocol is very similar to the one Stenström *et al.* propose in [20] which our adaptive protocol relies on. Therefore, we believe that our proposed adaptive competitive-update protocol fairly easy could be incorporated in bus-based systems.

In this study we simulated a system with only 16 processors. As we scale the system to a larger number of processors, we expect the latencies to be longer and it will be more important to keep the network bandwidth requirements at a low level. Therefore we believe that the benefits of our adaptive protocol increase with the system size, especially for applications that exhibit migratory sharing. The study of cache invalidation patterns by Gupta and Weber [10] shows that migratory sharing is independent of system size, at least in the range 8 to 32 processors which they use in their study.

In the Scalable Coherent Interface (SCI, IEEE standard P1596) [11] pair-wise sharing is discussed. Pair-wise sharing shows up when exactly two processors share the same memory block. In the SCI there is an optimization in the coherence protocol for this case which allows the two processors to pass the block between them without interaction with the home memory module. This mechanism is similar to the approach taken by our last-last-writer pointer. However, they do not use the mechanism to resort to competitive-update mode. Further, the SCI optimization does only work when the same two processors pass a block between them, not when a block migrates around among three or more processors.

7. Conclusion

The focus of this study has been to reduce the coherence miss rate and the network traffic caused by the cache coherence protocol for shared-memory multiprocessors. In this paper we propose an adaptive competitive-update cache coherence protocol that dynamically switches between two modes on a per block basis: competitive-update mode and migratory-mode, i.e., the read-exclusive optimization according to [20]. This study compares the performance of this adaptive protocol and suggests that it performs better than a write-invalidate protocol due to reduced traffic and miss rates.

A previous study has shown that competitive-update protocols [9] — a hybrid between write-invalidate and write-update protocols — outperform write-invalidate protocols for a wide range of applications under relaxed memory consistency models. However, for applications that exhibit migratory sharing [10] competitive-update protocols may generate unnecessary traffic that can increase the read penalty, especially for networks with a low bandwidth.

To improve the performance of competitive-update protocols, we propose in this study to extend them with a migratory detection mechanism, previously published for write-invalidate protocols [20], that dynamically detects migratory blocks and handles them with read-exclusive requests. As a result, all global write actions for migratory blocks are eliminated. In addition, in this study the migratory detection mechanism is extended to detect when exactly two processors modify the same memory block alternately, e.g., as a result of false sharing. By detecting this sharing behavior, the adaptive protocol resorts to competitive-update mode and reduces the coherence miss rate accordingly. We have also shown that the migratory detection mechanism adds only low extra complexity to the competitive-update protocol.

Based on program-driven simulations of a detailed multiprocessor architectural model, we find that the miss rates are reduced by up to 71% under the adaptive protocol as compared to a write-invalidate protocol. For applications with a high degree of migratory sharing the network traffic is reduced by up to 26% under the adaptive protocol as compared to a write-invalidate protocol and by up to 62% as compared to a competitive-update protocol. For applications with little or no migratory sharing, the adaptive protocol preserves almost the same low miss rates as the competitive-update protocol.

This study shows that by adding a simple mechanism for detection of migratory sharing the network traffic under competitive-update protocols is significantly reduced, and as a result, competitive-update protocols are shown to outperform write-invalidate protocols for networks with low bandwidths.

Acknowledgments

We would like to thank Lars Jönsson for implementing the basic adaptive protocol in our simulator as a part of his Master's thesis. This research has been funded in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under the contract number P855.

References

[1] Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[2] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenström, P. The CacheMire Test Bench — A flexible and effective approach for simulation of multiprocessors. *Proc. 26th Annual Simulation Symposium* IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 41-49.

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

[4] Cox, A. L., and Fowler, R. J. Adaptive cache coherency for detecting migratory shared data. *Proc. 20th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 98-108.

[5] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Stenström, P. The detection and elimination of useless misses in multiprocessors. *Proc. 20th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 88-97.

[6] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proc. 17th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 15-26.

[7] Gharachorloo, K., Gupta, A., and Hennessy, J. Performance evaluation of memory consistency models for shared-memory multiprocessors. *Proc. ASPLOS-IV.* ACM Press, New York, 1991, pp. 245-257.

[8] Goodman, J. R. Using cache memory to reduce processor-memory traffic. *Proc. 10th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 124-131.

[9] Grahn, H., Stenström, P., and Dubois, M. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems* **11**, 3 (June 1995), 247-271.

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

[11] IEEE. *IEEE - P1596 Draft Document, Scalable Coherent Interface Draft 2*.0. March 1992.

[12] Kroft, D. Lockup-free instruction fetch/prefetch cache organization. *Proc. 8th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 1981, pp. 81-87.

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

[14] Monier, L., and Sindhu, P. The architecture of the Dragon. *Proc. 30th IEEE Computer Society International Conference.* IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 118-121.

[15] Nilsson, H., and Stenström, P. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. *Proc. Parallel Architectures and Languages Europe (PARLE).* Lecture Notes in Computer Science, No. 817, Springer-Verlag, Berlin, 1994, pp. 363-374.

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

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

[18] Stenström, P., Dahlgren, F., and Lundberg, L. A lockup-free multiprocessor cache design. *Proc. 1991 International Conference on Parallel Processing.* CRC Press, Boca Raton, FL, 1991, Vol. I, pp. 246-250.

[19] Stenström, P., Joe, T., and Gupta, A. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. *Proc. 19th Annual International Symposium on Computer Architecture.* ACM Press, New York, 1992, pp. 80-91.

[20] Stenström, P., Brorsson, M., and Sandberg, L. An adaptive cache coherence protocol optimized for migratory sharing. *Proc. 20th International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 109-118.

HÅKAN GRAHN is an Assistant Professor of Computer Engineering at the University of Karlskrona/Ronneby in Sweden. He received a M.Sc. degree in Computer Science and Engineering in 1990 and a Ph.D. degree in Computer Engineering in 1995, both from Lund University. His main interests are computer architecture, shared-memory multiprocessors, cache coherence, and performance evaluation. He has authored and co-authored more than ten papers on these subjects. He received the Best Paper Award at the PARLE'94 (Parallel Architectures and Languages, Europe) conference and is a member of the Computer Society.

PER STENSTRÖM is a Professor of Computer Engineering with a chair in computer architecture at Chalmers University of Technology since 1995. He was previously on the faculty of Lund University where he also received his MS degree in Electrical Engineering and PhD degree in Computer Engineering in 1981 and 1990, respectively. Dr. Stenström's research interests are in computer architecture in general with a special emphasis on multiprocessor design and performance analysis as well as compiler optimization techniques. He has published more than 40 papers in these areas and has authored two textbooks on computer architecture and organization. As a visiting scientist, he has participated in major multiprocessor architecture research projects at Carnegie-Mellon, Stanford University, and University of Southern California. He is on the editorial board of the Journal of Parallel and Distributed Computing (JPDC) and has served on several program committees for computer architecture and parallel processing conferences including HPCA, ISCA, and ASPLOS. Dr. Stenström is a member of the IEEE and the Computer Society as well as ACM and the SIGARCH. For more information about Stentröm's current activities please refer to URL http://www.ce.chalmes.se/~pers.