In Parallel Architectures and Languages Europe (PARLE), pages 363-374, July 1994

An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic

Håkan Nilsson and Per Stenström

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

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 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 peeks for applications with extensive use of migratory objects. These traffic peeks 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 can outperform a write-invalidate protocol by reducing the miss rate and bandwidth need by as much as 71% and 26%, respectively.

1 Introduction

Using private caches in conjunction with a directory-based cache coherence protocol is a unified approach to reduce memory system latencies in large-scale multiprocessors [9]. Several cache coherence protocols have been proposed in the literature, both write-invalidate and write-update protocols.

Pure 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 [6] 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 [3] 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. However, 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* [4]. 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 cause unnecessary traffic that may 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 [2, 11] 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 handled according to the competitive-update policy.

Based on a detailed architectural simulation study using five parallel applications from the SPLASH benchmark suite [8] we find that competitive-update protocols extended with a simple migratory detection mechanism can reduce the read miss rate by as much as 71% as compared to a write-invalidate protocol. Our experimental results also show that the bandwidth requirements of the applications can be reduced by 26% as compared to a write-invalidate protocol for applications exhibiting migratory sharing. The read miss rate and traffic reductions can also help reducing the read penalty by 42% as compared to a write-invalidate protocol.

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 conclude the study in Section 6.

2 Cache Coherence Protocols and Migratory Sharing

In this section we first present our architectural framework. Then we discuss migratory sharing and how a write-invalidate protocol can be optimized for this type of access pattern. Finally, we briefly describe how the competitive-update protocol works and its performance limitations with respect to migratory objects.

2.1 The Architectural Framework

Our architectural framework is a cache coherent non-uniform memory access (CC-NUMA) architecture with a directory-based cache coherence protocol. The architecture consists of a number of processor nodes interconnected by a mesh network.

Each processor node consists of a processor with private caches, a local portion of the shared memory, and the network interface control, all connected by a local bus. Global cache coherence is supported by a directory-based full-map protocol [5]; each memory block associates a presence-flag vector indicating which nodes have a copy of the block. We 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 a coherence action.

In a CC-NUMA multiprocessor the processor stall times due to completion of memory accesses limit the performance of the whole system and stem mainly from two reasons; the read stall time arises 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) [3], and sufficient amount of buffering in the local node. This has been shown possible both under write-invalidate protocols [3] and under competitive-update protocols [6]. In order to achieve high performance we assume in this study the Release Consistency model.

2.2 Migratory Sharing and Write-Invalidate Protocols

Gupta and Weber classify data objects based on the access pattern they exhibit in [4]. *Migratory data objects* are manipulated by many processors but only a single processor at any given time. 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 [11]. The global reference stream to a migratory object can then be described by the following regular expression:

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

where 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.

Write-invalidate protocols rely on invalidation of remote copies to maintain coherence among cached copies of a shared memory block. In a write-invalidate protocol, the regular expression (1) results in the following coherence actions: If the block is not present in cache *i*, R_i results in a read-miss request and the block is loaded into cache *i* in a shared state. Upon the write request W_i to the block, the copy in memory is invalidated and cache *i* receives an exclusive (dirty) copy of the block.

When a subsequent 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* 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*. 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 optimization* [2, 11]. 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 [11], they found that a write-invalidate protocol extended with the migratory detection mechanism can reduce the number of global write requests by 96% for appli-

cations exhibiting migratory sharing. As a result, the network traffic is reduced by more than 20% for the studied applications with 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 ceases. A counter per cache line indicates the number of external updates since the last local access to the block. The network traffic is reduced as compared to a write-update protocol since only those copies regularly accessed are updated. More details on how competitive-update protocols are implemented and a detailed performance evaluation are found in [6]. However, we give a brief description of the protocol and some of the main results below.

On a local access to a block that resides in the cache, the counter associated with the block is preset to *C*. If a cache miss occurs, the block is fetched from memory, loaded into the cache, and the counter is preset to *C*. When the cache receives an update message from another processor, the cache controller checks the counter associated with the block. If the counter is zero the block is invalidated and an acknowledgment is returned to home indicating that the cache does not have a copy anymore, i.e., the updates to this cache ceases. Otherwise, the counter is decremented, the cache copy is updated, and an acknowledgment is sent to home indicating that the cache block is still valid.

The performance evaluation in [6] shows that competitive-update protocols can reduce the read penalty by 46% as compared to write-invalidate protocols, mainly as a result of a highly reduced number of coherence misses (up to 76% lower). The higher number of global writes under competitive-update protocols was shown not to offset the reduced read penalty under Release Consistency, assuming a mesh network as in the Stanford DASH multiprocessor [5].

Unfortunately, competitive-update protocols work suboptimally for migratory blocks since cached copies are often updated 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 a previously published migratory detection mechanism [11]. One would expect that such an extended protocol would reduce 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.

3 The Proposed Adaptive Protocol

In this section we identify the hardware mechanisms and the protocol extensions that incorporate the *migratory detection mechanism* into the competitive-update protocol, starting with a high-level view of the previously published detection mechanism [11].

3.1 A High-Level View of the Migratory Detection Mechanism

In order to identify migratory blocks, our *migratory detection mechanism* — which is conceptually the same as in [11] — detects the sequence W_i R_j W_j and classifies a memory block as migratory when W_j occurs. In [11], 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. If all blocks are classified as ordinary by default, home must detect when a block starts to be migratory. 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* 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 one *last-writer* pointer (LW) of size $\log_2 N$ bits, given N caches in the system, associated with each memory block 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. 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 modification by another processor. We 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.

3.2 A Competitive-Update Protocol with the Migratory Detection Mechanism

Given sequence (1) from Section 2.2, 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*. 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. 1, 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.

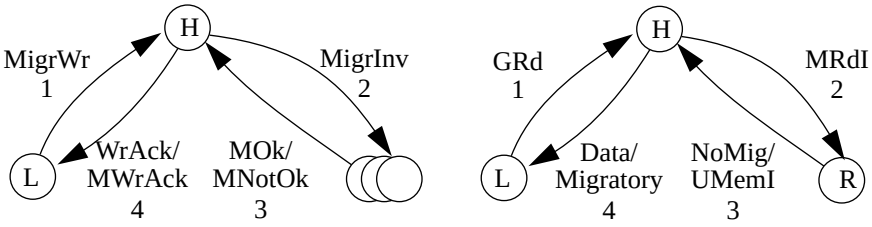

Fig. 1. Coherence actions for detection of migratory blocks (left) and coherence actions for read misses to migratory blocks (right).

Read-miss requests to migratory blocks are handled according to the right part of Fig. 1. 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 [11], 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 modi-

fied 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 An Enhanced Adaptive Protocol

In applications with false sharing between two processors, a problem can arise in the adaptive protocol described in Section 3.2. Consider the following reference sequence to a block, where a block is alternatively accessed by processor *i* and *j*:

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

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 Experimental Methodology

In this section, we present the simulation framework in which we have studied the effectiveness of the adaptive protocol according to Section 3. The simulation models are built on top of the CacheMire Test Bench [1]; 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. We model 16 processing nodes connected by a mesh network.

The organization of a processor node is shown in Fig. 2. The two-level cache hierarchy we simulate consists of a first-level cache (FLC) and a second-level cache (SLC) with associated write-buffers (16 entries each). The FLC is a 2 Kbyte write-through on-chip cache whereas the SLC is an infinite copy-back cache. Both caches are direct-mapped with a line size of 16 bytes and full inclusion is supported. The SLC is lockup-free [10] 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. Acquire and release requests are supported by a queue-based lock mechanism similar to the one implemented in the DASH multiprocessor [5]. The page size is 4 Kbyte and the pages are allocated to memory modules in a round-robin fashion.

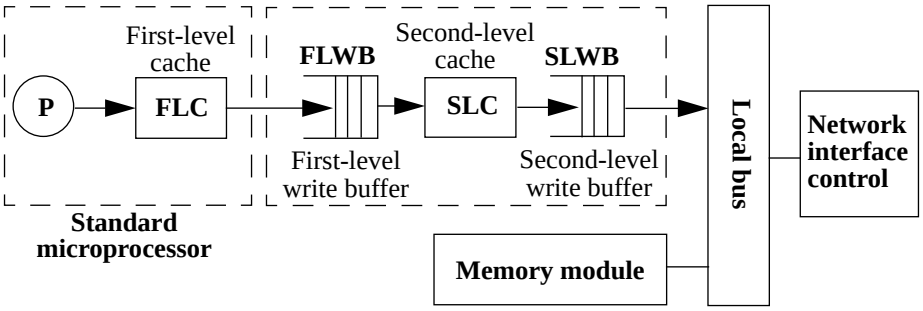

Fig. 2. The organization of a processor node.

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 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 conservative mesh implementation that is clocked at 33 MHz. We correctly model contention for all parts in the system. With these assumptions it takes 1, 4, and 28 pclocks to satisfy a processor read request from the FLC, the SLC, and from the local memory, respectively. A read miss serviced in home and remote takes 100 and 196 pclocks, respectively, assuming a contention-free system. (In our simulations, however, requests normally take a longer time as a result of contention.)

In order to understand the relative performance of the adaptive protocol, we use five scientific and engineering applications. Four of them are taken from the SPLASH suite (MP3D, Water, Cholesky, and PTHOR) [8]. The fifth application (Ocean) has been provided to us from Steven Woo of Stanford University. All programs are written in C using the ANL macros to express parallelism and compiled with `gcc` version 2.1 (optimization level `-O2`).

MP3D was run with 10 000 particles for 10 time steps. Cholesky used the `bcsstk14` matrix. Water was run with 288 molecules for 4 time steps. PTHOR used the RISC circuit and was run for 1000 time steps. Finally, Ocean used a 128x128 grid with the tolerance factor set to 10^{-7}. Previous studies [4, 11] have shown that MP3D, Cholesky, and Water have a high degree of migratory objects, while PTHOR and Ocean have many producer-consumer objects.

5 Experimental Results

In this section we evaluate performance of the enhanced adaptive protocol described in Section 3.3, henceforth referred to as AD+. A previous study [6] showed 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 assuming a 100 MHz mesh. However, for the 33 MHz mesh we use in this study, a competitive-update protocol may suffer from a higher network traffic.

In Fig. 3 the normalized execution times of the write-invalidate (WI), the competitive-update (CU), and the enhanced adaptive (AD+) protocols are presented for the five applications under study. Three bars are associated with each application and correspond to WI, CU, and AD+, 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. In our measurements, we have assumed a competitive threshold of 4.

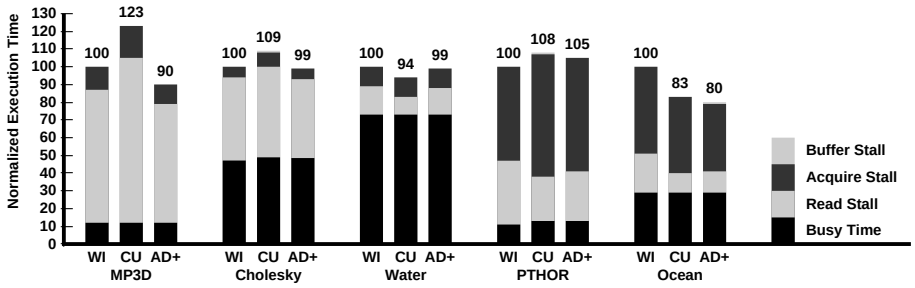

Fig. 3. Normalized execution times for WI, CU, and AD+.

Comparing WI and CU, we observe in Fig. 3 that both MP3D and Cholesky have longer read stall times under CU than under WI, mainly as a result of network contention. In Table 1, the bandwidth requirements for each application is presented, calculated as the network traffic divided by the execution time and normalized to the bandwidth requirement of WI for each application. We observe that WI requires less bandwidth than CU for all applications. The large number of unnecessary updates to migratory objects results in a dramatic difference in the bandwidth needed for MP3D and Cholesky. For MP3D we see that CU requires 51% more bandwidth than WI and for Cholesky CU requires 86% more bandwidth than WI.

In Fig. 3, we also observe that for PTHOR the acquire stall time is significantly higher under CU than under WI. 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 delays the issuance 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 next.

Table 1: Read miss rates and normalized bandwidth requirements of the applications.

Application	Read miss rates in percentage (Cold + Coherence)			Normalized bandwidth requirements (WI = 100%)	
	WI	CU	AD+	CU	AD+
MP3D	2.4 + 13.0	2.4 + 11.1	2.4 + 12.7	151%	74%
Cholesky	1.5 + 0.43	1.4 + 0.37	1.4 + 0.40	186%	90%
Water	0.07 + 1.2	0.07 + 0.98	0.07 + 1.0	197%	74%
PTHOR	3.3 + 3.6	3.0 + 0.95	3.0 + 1.3	121%	116%
Ocean	0.04 + 0.73	0.04 + 0.18	0.04 + 0.18	149%	156%

We observe by comparing the execution times for AD+ and WI that overall, AD+ manages to outperform WI for all applications but one (PTHOR). The execution times have decreased under AD+ by between 1% and 20%. For MP3D and Cholesky the main reason is that the read penalty has been reduced by 11% and 4%, respectively, thanks to lower contention in the network. Furthermore, 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 because of contention for locks. 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 23% as compared to WI. 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 which means that the read penalty is not affected by network contention under WI.

In Table 1, we also show the read miss rates decomposed in the cold and coherence miss components for each application. The read miss rates are the number of processor reads that miss in both the FLC and the SLC divided by the number of processor reads. As expected, for MP3D the miss rates of AD+ are almost the same as under WI. The miss rates for Cholesky and Water are 7% and 11% lower, respectively, under AD+ than under WI even though most data objects are migratory. However, there are some objects that are not migratory. Those objects resort to competitive-update, and thus, an overall lower coherence miss rate is encountered. For applications with marginal migratory sharing (PTHOR and Ocean) the read miss rates are reduced under AD+ by 38% and 71%, respectively, as compared to WI. Hence, AD+ has preserved the low read miss rates of CU for applications with little or no migratory sharing.

As for the traffic reduction by AD+, we see in Table 1 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 almost the same under AD+ and CU for applications with little or no migratory sharing (PTHOR and Ocean).

We have also evaluated the benefits of AD+ as compared to the basic adaptive protocol described in Section 3.2, henceforth referred to as AD, which uses a single LW pointer per memory block. The only application where AD and AD+ differ in performance is Ocean, which has a non-negligible degree of false sharing. Simulation results show that AD has 38% higher read penalty and 136% more read misses than AD+. As a result, the execution time is 15% longer under AD than under AD+. A more thorough analysis is presented in [7].

To summarize, AD+ appears to be a better default policy than WI and CU because 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.

6 Concluding Remarks

To improve the performance of competitive-update protocols, we have proposed to extend them with a previously published migratory detection mechanism [11] 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, 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 marginal complexity to the competitive-update protocol.

Based on program-driven simulations of a detailed multiprocessor architectural model, we have shown that for competitive-update protocols it is possible to reduce the read miss rates and the bandwidth requirements by as much as 71% and 26%, respectively, 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 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 increases with the system size, especially for applications that exhibit migratory sharing. The study of cache invalidation patterns by Gupta and Weber [4] shows that migratory sharing is independent of system size, at least in the range 8 to 32 processors which they use in their study.

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, such adaptive competitive-update protocols are shown to outperform write-invalidate protocols for networks with low bandwidths.

Acknowledgment

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 work was supported by the Swedish National Board for Industrial and Technical Development (Nutek) under the contract number 9001797.

References

1. M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, "The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors", In *Proceedings of the 26th Annual Simulation Symposium*, pp. 41-49, March 1993
2. A.L. Cox and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data", In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, May 1993
3. K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors", In *Proceedings of ASPLOS IV*, pp. 245-257, April 1991
4. A. Gupta and W-D. Weber, "Cache Invalidation Patterns in Shared-Memory Multiprocessors", *IEEE Transaction on Computers*, 41(7), pp. 794-810, July 1992
5. D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor", *IEEE Computer*, 25(3):63-79, March 1992
6. H. Nilsson, P. Stenström, and M. Dubois, "*Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models*", Technical Report, Dept. of Computer Engineering, Lund University, Sweden, July 1993
7. H. Nilsson and P. Stenström, "*Evaluation of Adaptive Update-Based Cache Protocols*", Technical Report, Dept. of Computer Engineering, Lund University, Sweden, March 1994
8. J-P. Singh, W-D. Weber, and A. Gupta. "SPLASH: Stanford parallel applications for shared-memory", Computer Architecture News, 20(1):5-44, March 1992
9. P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, 23(6):12-24, June 1990
10. P. Stenström, F. Dahlgren, and L. Lundberg, "A Lockup-free Multiprocessor Cache Design", In *Proceeding of 1991 International Conference on Parallel Processing*, Vol. I, pp. 246-250, August 1991
11. P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing", In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 109-118, May 1993