An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing

Per Stenström, Mats Brorsson, and Lars Sandberg
Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden

Abstract

Parallel programs that use critical sections and are executed on a shared-memory multiprocessor with a write-invalidate protocol result in invalidation actions that could be eliminated. For this type of sharing, called migratory sharing, each processor typically causes a cache miss followed by an invalidation request which could be merged with the preceding cache-miss request.

In this paper we propose an adaptive protocol that invokes this optimization dynamically for migratory blocks. For other blocks, the protocol works as an ordinary write-invalidate protocol. We show that the protocol is a simple extension to a write-invalidate protocol.

Based on a program-driven simulation model of an architecture similar to the Stanford DASH, and a set of four benchmarks, we evaluate the potential performance improvements of the protocol. We find that it effectively eliminates most single invalidations which improves the performance by reducing the shared access penalty and the network traffic.

1 Introduction

In order for shared-memory multiprocessors to achieve a high performance, memory system latency and contention must be kept as low as possible. A viable solution to this problem has been to attach private caches to each processing node and maintain cache coherence using a directory-based write-invalidate protocol. Notable examples of real implementations of large-scale multiprocessors that exploit this technique are the Stanford DASH [12], the MIT Alewife [1], the SICS Data Diffusion Machine (the DDM) [9], and the Kendall Square Research's KSR1 [2].

Write-invalidate protocols maintain cache coherence by invalidating copies of a memory block when the block is modified by a processor. The advantage of this is that at most the first write, in a sequence of writes to the same block with no intervening read operations from other processors, causes global interaction. Consequently, write-invalidate protocols perform fairly well for a broad range of sharing patterns. However, there exist common sharing patterns for which all invalidations could have been entirely avoided. A notable example is the invalidation overhead associated with data structures that are accessed within critical sections. Typically, processors read and modify such data structures one at a time. Processors that access data this way cause a cache miss followed by an invalidation request being sent to the cache attached to the processor that most recently exited the critical section. If the cache coherence protocol were aware of this sharing pattern, it would be possible to merge the invalidation request with the preceding read-miss request and thus eliminate all explicit invalidation actions. This sharing behavior, denoted *migratory sharing*, has been previously shown to be the major source of single invalidations by Gupta and Weber in [8].

Eliminating invalidation requests can help performance in many important ways. First, if access requests cannot overlap invalidation requests due to memory consistency model or implementation constraints [6], the access penalty is reduced by reducing the number of global invalidation requests. Second, the network traffic is reduced which, as a secondary effect, may reduce the read and write penalty due to less network contention. Consequently, eliminating the number of invalidation requests may improve the performance significantly.

In this paper, we propose an implementation of an adaptive write-invalidate protocol that effectively eliminates most invalidation requests associated with migratory sharing. The protocol dynamically detects whether a memory block exhibits migratory sharing or not. For blocks deemed migratory, the invalidation request is merged with the preceding read-miss request and for other blocks, it maintains coherence according to the default write-invalidate policy. In addition, the protocol can dynamically, on a per block basis, switch between these operating modes, would the block change sharing behavior. As a case-study, we show that our protocol is a simple extension of a write-invalidate protocol by presenting the modifications needed for a state-of-the-art write-invalidate protocol, in essence the directory-based protocol of the Stanford DASH [11].

To validate the correctness of the protocol and evaluate its performance, we have implemented and evaluated it using a detailed program-driven simulation model of a DASH-like architecture and a set of four benchmarks, of which three are taken from the SPLASH suite [14]. We have found that by eliminating the invalidation requests to

0884-7495/93 $3.00 © 1993 IEEE

migratory blocks, performance can be improved due to less access penalty and network traffic. We show that these factors can improve the performance significantly for high-performance multiprocessors.

The organization of the rest of the paper is as follows. Since the adaptive protocol can be applied to most write-invalidate protocols, we provide in Section 2 a high-level description of the coherence optimization it provides and how it detects migratory sharing. As a base for our implementation and performance study, we use the Stanford DASH protocol. In Section 3, we describe how the DASH protocol can be extended to adaptively detect and optimize migratory sharing. Sections 4 and 5 present the experimental results starting with the architectural assumptions and the benchmarks in Section 4. We finally generalize our findings in Section 6 and conclude the paper in Section 7.

2 The Adaptive Protocol: A High-Level View

In this section, we first identify the type of migratory sharing that incurs invalidation overhead in write-invalidate protocols in Section 2.1. Then in Section 2.2, we present a high-level description of an adaptive protocol that detects such sharing and eliminates its overhead.

2.1 Migratory Sharing

Gupta and Weber classify data structures based on the invalidation pattern they exhibit [8]. According to their definition, data structures manipulated by only a single processor at any given time are called *migratory objects*. Typically, such sharing occurs when a data structure is modified within a critical section. Protecting modifications of shared objects by locks is important to eliminate data races. Therefore, many parallel languages, such as e.g. Modula-2 and Ada, use critical sections (or monitors) as the recommended mechanism to support shared data access. Consequently, reducing access penalties incurred by migratory objects is important.

Assuming a write-invalidate protocol, all blocks that correspond to a migratory object being modified by a processor end up dirty in the processor's cache. When a subsequent processor modifies the migratory object, it causes a single invalidation for each block that belongs to the object. If a migratory block is read and then modified by each processor, the first read access will cause a miss and the first write access will cause a single invalidation, both being sent to the cache associated with the processor that previously modified the block. Clearly, the invalidation request could be merged with the preceding miss request.

To formally define all access sequences that cause such invalidation overhead due to migratory sharing, we use the symbols R_i and W_i to denote a read and write access to a memory block by processor i. The following regular expression defines all such sequences:

$$(\mathbf{R_i})(R_i)^*(\mathbf{W_i})(R_i / W_i)^*(\mathbf{R_j})(R_j)^*(\mathbf{W_j})(R_j / W_j)^* \ldots \quad (1)$$

In the above regular expression a '*' succeeding a string designates a string of arbitrary length including the empty string and a 'I' denotes the OR-operator. In this sequence, there is one global read followed by a global write access by the same processor before this processor relinquishes exclusive access of the block for another processor. These accesses are marked with boldface.

2.2 Adaptive Detection and Optimization

As a base for the adaptive protocol, we assume a shared-memory multiprocessor that contains a number of processors with associated caches. To maintain coherence, there is an explicit *home* associated with each memory block that keeps track of the global coherence state of the block (e.g. uncached, shared, or dirty). Home dynamically decides whether a block is *migratory*, meaning that it does adhere to the sharing pattern in (1) above, or *ordinary*, meaning it does **not** adhere to (1). Below we describe the coherence actions for migratory and ordinary blocks and the detection algorithm to choose between these coherence policies.

For ordinary blocks, cache coherence is maintained by a write-invalidate protocol as follows. If a read access causes a miss in the cache, a *read-miss request* is sent to home. Depending on whether the memory block is valid or not, either home or the cache that keeps the dirty copy responds with a copy of the block to the local cache. In either case, home will keep a valid copy in state shared. If the local cache copy is shared or invalid, a write access causes an invalidation request, called *read-exclusive request*, to be sent to home. Home is responsible for invalidating all copies of the block.

For migratory blocks, when home receives a read-miss request, it converts this request into a read-exclusive request and forwards it to the cache that has the block in state dirty. This cache gives up its copy for the requesting cache that will load the block in state dirty. As a result, a subsequently issued write access by the requesting processor can be carried out locally and all explicit invalidation actions have been eliminated.

Since all memory blocks are tagged ordinary by default, home can detect migratory sharing based on the fact that it receives the global sequence of read-miss (Rr_i) and read-exclusive (Rxq_i) requests from each processor i for a block. If an ordinary block starts exhibiting migratory sharing, the global sequence is: $Rr_i\ Rxq_i\ Rr_j\ Rxq_j\ Rr_k\ Rxq_k \ldots$ By always keeping track of the identity of the processor that most recently wrote to the block, say i, a block can be nominated as migratory at a point when home receives a read-exclusive request from j given that:

(1) the read-exclusive request comes from a different processor ($j \neq i$) and (2) the number of copies is exactly two.

In summary, the notion of home is important for implementing the adaptive protocol because home sees all read-miss and read-exclusive requests for the block. Cache-coherent NUMA (CC-NUMA) machines [16] constitute an example where home is the memory in which the block is allocated. We will next consider the detailed implementation of the adaptive optimization in an example CC-NUMA machine — the Stanford DASH.

3 An Implementation Study: Stanford DASH

The usefulness of the coherence optimization of migratory blocks as described in the previous section is dictated by the extra hardware complexity required by the protocol and the performance improvements that can be obtained. As a base to address these issues, we have chosen the Stanford DASH protocol [11, 12] because it has been implemented and sufficiently tested and documented. In Section 3.1, we review the Stanford DASH protocol with respect to the actions taken for migratory sharing. In Section 3.2, we describe how these coherence actions are optimized by the adaptive protocol extension. Finally, in Sections 3.3 and 3.4, we present how we extend the DASH protocol to detect and optimize coherence actions for migratory blocks.

3.1 The DASH Write-Invalidate Protocol

In Figure 1, we show the organization of each node. To simplify, we assume that each node contains a single processor. The processor with its associated cache is connected to the local memory module by a local bus. Shared memory is distributed across the processing nodes, which are interconnected by two two-dimensional wormhole-routed mesh networks — one for requests and one for replies.

Cache coherence at the system level is maintained by a directory-based write-invalidate protocol by a directory for each memory module that keeps track of the global state of all memory blocks in this memory module (the home of its blocks). Three global states are associated with each memory block: *Uncached* (not cached by any other node than home), *Shared-Remote* (valid copies exist in other nodes), and *Dirty-Remote* (the block is modified in some other node's cache). Furthermore, if the global state is not Uncached, home keeps track of the nodes that have a copy, the *sharing list*, by a presence-flag vector containing the same number of bits as nodes.

Similarly, each cache keeps track of the local state of its copy by three states: *Invalid* (nonexistent in the cache), *Shared* (valid block in this cache and possibly in other nodes' caches), and *Dirty* (the only copy exists in this cache). We now review the coherence actions taken by the DASH protocol by denoting the issuing node *local* and a

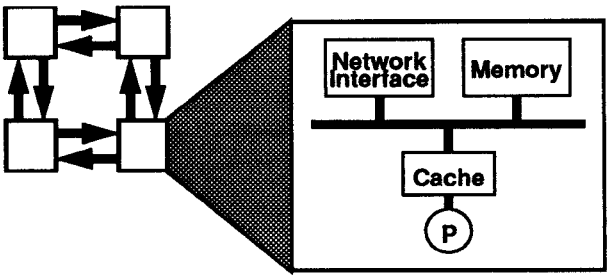
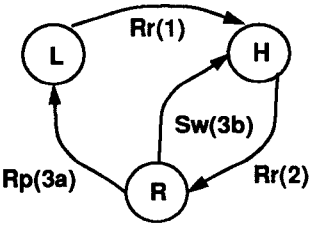


Figure 1: Processing node organization.

2(a)



2(b)

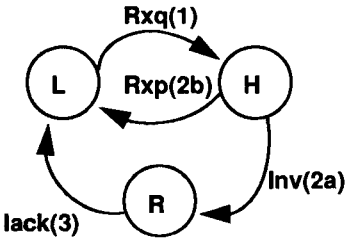


Figure 2: Example coherence actions for DASH. (a) Read-miss and (b) read-exclusive request actions associated with migratory sharing.

node other than the local or the home node *remote*. For simplicity, we will assume that local, home, and remote are distinct nodes.

Processor reads are satisfied by the local cache if the local state is Shared or Dirty. If the state is Invalid, or if there is a tag mismatch, local (L) sends a read-miss request (Rr in Figure 2(a)) to home (H). If the global state of the block is Uncached or Shared-Remote, a copy is returned to the local node and home updates the sharing list to incorporate the new keeper. If the global state is Dirty-Remote (as in Figure 2(a)), home forwards the read request to the node that keeps the dirty copy. This node, the remote node (R), returns the block to local (Rp) and to the home node (Sw) and changes the state of its copy to Shared. The final global state is Shared-Remote.

Moving on to the actions associated with processor writes, we first note that if the local state is Dirty, they can be carried out locally. If the state is Shared or Invalid,

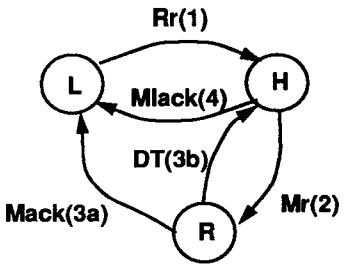


Figure 3: Coherence actions for migratory blocks.

local sends a read-exclusive request to home to acquire ownership as depicted in Figure 2(b) as Rxq. If the global state is Uncached, an exclusive copy is returned to local (Rxp). If the global state is Shared-Remote, however, home also sends invalidations to all remote nodes that have a copy of the block (Inv in Figure 2(b)). When receiving such an invalidation, each node changes the state of its copy to Invalid and sends an acknowledgment to the local node (Iack). The final global state is Dirty-Remote.

In the DASH protocol, coherence of all blocks are maintained according to the actions described in this section. In the adaptive protocol extension we evaluate in this paper, ordinary blocks are handled according to the DASH protocol. How migratory blocks are handled is described next.

3.2 Coherence Maintenance of Migratory Blocks

Recalling the regular expression for global accesses to migratory blocks (1) in Section 2.1, we note that when a new processor starts to access the block, a dirty copy exists in the cache associated with the processor that most recently relinquished exclusive access to the block. Consequently, the read-modify-write access to the migratory block results in a read-miss request (according to the actions in Figure 2(a)) followed by a read-exclusive request (according to the actions in Figure 2(b)) resulting in a single invalidation assuming the DASH protocol.

The adaptive protocol converts these two transactions into a single one that takes place at the time a read-miss request is sent to home according to Figure 3. As in Figure 2(a), local sends a read-miss request to home. Then home forwards the read-miss request to the remote node (depicted Mr in Figure 3). Unlike the actions in Figure 2(a), however, remote gives up ownership of the block by sending its copy to local (Mack). It also notifies home about the ownership change with a request (DT). When local receives the reply message from remote, the cache is filled and the processor is restarted. As a result, the processor stall-time to service the read request, counted in network hops, is the same as in Figure 2(a). However, the local node is not allowed to replace the block until home has updated its directory. This is acknowledged by MIack

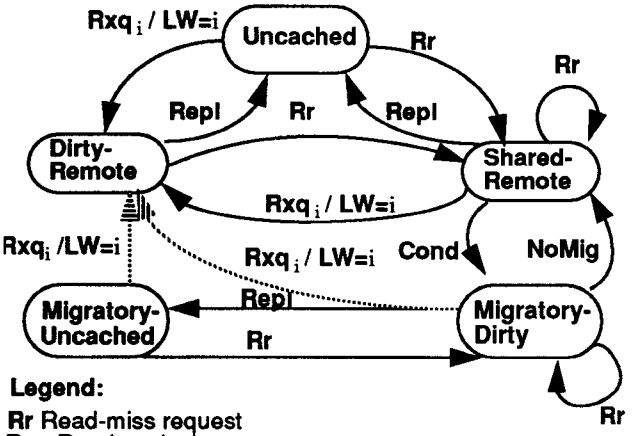


Figure 4: State-transition graph for detection of migratory blocks.

in Figure 3. The extra acknowledgment is needed to avoid corrupting the directory.

The performance improvement from this optimization is due to a reduction of write accesses causing global actions, which may reduce the write penalty under sequential consistency [10] and network traffic. The latter can reduce read and write penalty because of a reduction of network contention. In Section 5, we will quantify these effects.

3.3 The Detection Algorithm

The finite-state machine that keeps track of the global state of each block in DASH consists of three states: Uncached, Shared-Remote, and Dirty-Remote. Transitions between these states occur as a result of read-miss, read-exclusive, and replacement requests for ordinary blocks according to Figure 4. In order to detect and keep track of migratory blocks, we have augmented the finite-state machine with two states: *Migratory-Dirty* and *Migratory-Uncached.*

To detect migratory blocks, we recall from Section 2.2 that we have to keep track of the identity of the processor that most recently modified the block. This is done by associating a pointer, the *last-writer* pointer (LW), with each memory block which is updated at each transition to Dirty-Remote in Figure 4 (LW=i). In addition, we must detect when there are exactly two copies. In the DASH protocol this is known from the presence-flag vector.

According to condition Cond in Figure 4, an ordinary block is now nominated as migratory at the time home receives a read-exclusive request from processor *i* (Rxq_i in Figure 4) and (1) the number of cached copies is two (N==2), and (2) the writing processor is not the same as the previous writer (LW != *i*). Requirement (1) prevents a state transition as a result of intervening read-miss

requests from other processors while the global state is Shared-Remote. Otherwise, the following sequence could be detected as migratory: $Rxq_i\ Rr_j\ Rr_k\ Rxq_j$. Requirement (2) prevents a producer-consumer sequence, such as $Rxq_i\ Rr_j\ Rxq_i\ Rr_j$, from being detected as migratory. Note also that we must associate a valid bit with the last-writer pointer, which is initially reset. It must also be reset as soon as the size of the sharing list exceeds two to prevent an erroneous transition to Migratory-Dirty as a result of replacements such as in: $Rr_i\ Rxq_i\ Rr_j\ Rr_k\ Repl_k\ Rxq_j$, where $Repl_k$ denotes that the block is replaced in cache k.

Once the block has been nominated as migratory, the memory controller will handle all subsequent read-miss requests according to the actions depicted in Figure 3; a read request will result in an ownership to be obtained. Consequently, there is only a single copy in the system and invalidations for migratory blocks are eliminated. To avoid having to re-detect a block as migratory when it is written back to memory as a result of replacement, we have added the state Migratory-Uncached.

3.4 Adaptivity to Alterations in Sharing Behavior

To cope with alterations in the sharing behavior, the protocol can perform transitions between the write-invalidate and the migratory policy. We study below a few situations our protocol supports.

The fact that the memory controller sees only read-miss requests for migratory blocks means that it cannot detect if a block starts to be read-only. For example, if processors i and j alternately read from a migratory block, the memory controller will see the following sequence of read requests: $Rr_i\ Rr_j\ Rr_i\ Rr_j$... Clearly, the block will ping-pong back and forth between cache i and j. To avoid this, we have added a local cache state denoted *Migrating* which is used as follows. When processor i reads the block, the initial local cache state is Migrating instead of Dirty. A subsequent write to the block results in a transition to Dirty without any global action being taken. However, if the cache receives a migratory read request (Mr in Figure 3) and the local state is Migrating, then instead of giving up ownership, it performs the same actions as in Figure 2(a) by notifying the memory controller with a NoMig request (see Figure 4). The block is also written back to home and the global state is Shared-Remote. The block is considered as ordinary from this point.

We have assumed that the first access to a migratory block by a processor is always a read. If it is a write, however, should we still regard it as migratory, or should we make a transition to Dirty-Remote? As a default policy, we still consider the block as migratory but we have also evaluated the heuristic of making a transition to Dirty-Remote when home receives a read-exclusive request, which is depicted with dashed arrows in Figure 4.

In summary, the adaptive extension to the DASH protocol consists of a pointer per block with $\log_2 N$ bits assuming N nodes, two global and one local cache state in addition to the mechanisms that are already there. In the next two sections, we will see what this extra hardware complexity can buy us in terms of increased performance.

4 Evaluation Methodology

To validate the correctness of the adaptive cache coherence protocol and its potential performance benefits, we have used a simulation methodology using detailed architectural models in conjunction with a suite of parallel applications. We next present the simulation environment, the architectural parameters, and the benchmark programs used.

4.1 Simulation Environment

The simulation platform used is the CacheMire test bench [4] — a program-driven simulator of multiple SPARC processors on top of which it is possible to run parallel applications written in C using the Argonne National Laboratory's macro package to express parallelism [3]. The test bench consists of two parts: (i) a functional simulator and (ii) an architectural simulator. The functional simulator generates memory references that are performed in the architectural simulator. In order to maintain a correct interleaving of memory references, the architectural simulator maintains the global time and delays the processors according to its timing model. As a result, a correct interleaving of events in the architectural model is maintained. This is in contrast to e.g. trace-driven simulation, where the memory reference trace is not affected by timing.

4.2 Architectural Model and Assumptions

The basic architectural model is similar to the Stanford DASH and we show the overall organization of the architecture in Figure 1. Although we present results for architectural variations, we will only focus on the default assumptions in this section.

We assume a 16 node configuration interconnected by two 4x4 wormhole-routed meshes — one for requests and one for replies. Each processing node contains a memory module and a 64 Kbyte, direct-mapped, copy-back cache with a line size of 16 bytes. As for the memory allocation, we allocate shared data pages in a round-robin fashion with the least significant bits of the virtual page number designating the node number. The page size is 4 Kbytes. For simplicity, we only model shared data references; instruction and private data references are assumed to always hit in the on-chip processor caches.

The cache and the memory module are connected by a 128-bit wide split-transaction bus which also provides a connection to the network interface. The network interface

Table 1: Latency numbers (1 pclock = 10 ns).

Latency for Read and Write Requests	Time
Hit in Cache	1 pclock
Fill from Local Memory	22 pclocks
Fill from Remote (2-hop)	54 pclocks
Fill from Remote (3-hop)	73 pclocks
Read-Exclusive Request to Remote (2-hop)	51 pclocks
Read-Exclusive Request to Remote (3-hop)	70 pclocks

routes requests and replies to the corresponding mesh network. It also keeps track of all outstanding requests from the node by means of a mechanism similar in function to the remote-access cache (RAC) in DASH [11].

The two wormhole-routed meshes have a link width of 16 bits. The node fall-through time corresponds to three pipeline stages: arbitrate, route, and send. Infinite buffers are associated with each of the four inputs (X+1 X-1, Y+1, and Y-1) to each network router.

As for the timing model parameters, we assume that the processors are clocked at 100 MHz (1 pclock = 10 ns) and that the cache access time is 10 ns. Moreover, the local bus is assumed to be clocked at 50 MHz — it takes 20 ns for arbitration and 20 ns for the bus transfer. The memory cycle time is assumed to be 100 ns including buffering. Finally, we assume a fairly aggressive mesh implementation that is synchronously clocked at 100 MHz which results in a peak bandwidth of 400 Mbytes/sec out from and into each node.

We model contention correctly at the memory modules, the local buses, and the mesh networks. In Table 1, we list the latencies for processor reads and writes depending on where in the memory hierarchy they are serviced. A 2-hop remote latency means that the request is serviced in two network traversals, e.g. a read to a block that is clean at home and home is not the local node. The remote latencies we model depend on where in the mesh the interacting nodes are situated and on contention. For the latency numbers in Table 1, however, we assume no contention and an average distance of 2.67 links for a network traversal between two arbitrary nodes (the mean distance in a 4x4 mesh).

As far as the memory consistency model is concerned, we assume sequential consistency (SC) [10]. We implement SC by stalling the processor on every read-exclusive request to a cache copy that is Shared or Invalid until the write has been performed. Finally, for simplicity we handle synchronization requests (locks and barriers) ideally with a single-cycle delay outside the architecture model because we feel that their implementations are orthogonal issues to the focus of this study.

4.3 Benchmark Programs

In order to study the relative performance of the DASH protocol with and without the adaptive extension, we have used a set of four scientific applications developed at Stanford University of which three (MP3D, Cholesky, and Water) are from the SPLASH suite [14]. A summary of these applications is given in Table 2. We picked this set of applications to show different aspects of migratory sharing.

Table 2: Benchmark programs.

Benchmark	Description
MP3D	Particle-based wind-tunnel simulator
Cholesky	Cholesky factorization of sparse matrices
Water	Water molecular dynamics simulation
LU	LU decomposition of dense matrices

MP3D was run with 10K particles for 10 time steps. Cholesky was run using the `bcsstk14` benchmark matrix. Water was run with 288 molecules for 4 time steps, and finally, LU uses a 200x200 matrix.

The applications were compiled using `gcc` (version 2.0) with the optimization level -O2. Statistics acquisition is started when the applications enter the parallel section to study steady-state behavior.

5 Experimental Results

In this section, we study the performance improvements provided by the adaptive protocol. In Section 5.1, we compare the execution times of the benchmarks for the adaptive and the write-invalidate protocol by analyzing the occurrence of migratory sharing. Then we move on to see how the adaptive protocol can help performance by reducing the network traffic in Section 5.2. Section 5.3 deals with the impact of cache size on performance, and finally, in Section 5.4, we study the stability of the detection algorithm.

5.1 Performance of Write-invalidate and Adaptive

The performance improvement of the adaptive protocol is dictated by the occurrence of migratory objects in the applications. In Figure 5, we show the execution time under the adaptive protocol (AD) normalized to the execution time without the adaptive extension (W-I).

We observe that the adaptive protocol performs consistently better than write-invalidate by examining the execution-time ratio (ETR) of W-I relative to AD. For example, the adaptive protocol results in 54% better performance (ETR=1.54) for MP3D. The reason for this is that the processors have to stall less as a result of invalidation requests since the adaptive protocol reduces the number of read-exclusive requests. To see this, we have broken down

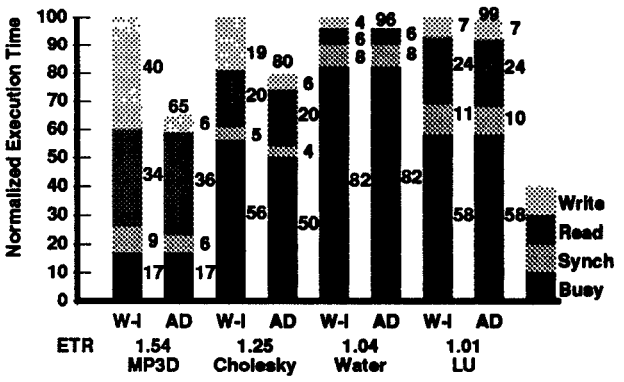


Figure 5: Relative performance of W-I and AD.

the execution time into the contributions due to busy time, synchronization (time waiting for a lock or at a barrier), read, and write stall-time from the bottom to the top. For example, the busy time for W-I running MP3D is 17%, while the synchronization stall-time is 9%. We see that the write stall-time has been significantly reduced for most applications under the adaptive protocol. We therefore review each application below with respect to the occurrence of migratory sharing.

In a previous work, Gupta and Weber [8] studied the invalidation pattern for two of the applications we use. They observed for MP3D and Water that more than 98% of the read-exclusive requests resulted in single invalidations. In MP3D, most accesses to shared data are caused by reading and modifying the particle and space-array entries. Even though the modifications are not protected by locks, they behave as migratory because a modification by a processor follows closely after the read access. In Table 3, we show the reduction of read-exclusive requests for each application. As expected, we see a reduction of read-exclusive requests for MP3D by as much as 87%.

Cholesky performs factorization using supernodal modifications. Supernodes are groups of columns with a similar structure. The computation is mastered by a global task queue that keeps track of all supernodal modifications that are to be done. Typically, a processor pulls a supernode off the task queue and performs modifications on other super-nodes which are protected by locks. The migratory sharing that shows up is due to the task queue and to the supernodal modifications themselves. As expected, the number of read-exclusive requests is reduced by 69% (see Table 3), which results in 25% better performance for AD according to Figure 5.

Since Cholesky dynamically schedules work among the processors, there is a discrepancy in the busy time for W-I and AD. It should therefore be noted that not all of the performance improvements are due to a write-stall reduction.

In Water, the molecule array is statically split among processors. Each processor calculates the pair-wise interaction between its molecules and those of others. These modifications are protected by locks and result in migratory sharing. As a result, virtually all read-exclusive requests are eliminated by the adaptive protocol (a 96% reduction). Surprisingly, the execution time is reduced by only 4%. As we easily can make out from Figure 5, the reason is that there is not more to gain — the write stall-time is 4%.

Table 3: Reduction of read-excl. requests and traffic.

Application	Read-excl. Reduction	Traffic Reduction
MP3D	87%	32%
Cholesky	69%	22%
Water	96%	31%
LU	5%	1%

In LU there are virtually no migratory objects, and consequently, no performance improvement. However, LU demonstrates that the adaptive protocol does not impact adversely on the performance as a result of erroneous detections.

In summary, we have seen that the adaptive protocol is successful in reducing the number of read-exclusive requests to migratory blocks. The execution-time reduction is due to the reduced write penalty associated with the sequential consistency model.

5.2 Network Traffic Reduction Effects

The adaptive protocol can also reduce the access penalty by reducing network contention as a result of traffic reduction which is the focus of this section. The reduced traffic is due to the fact that the messages associated with a read-miss request (according to Figure 2(a) in Section 3.1) and a read-exclusive request (according to Figure 2(b)) under W-I have been replaced by the messages according to Figure 3 under AD. To get a feel for the traffic reduction, we review how many bits these messages occupy.

Requests and replies contain the identity of the issuing and receiving node (assuming 16 processors this corresponds to 4 + 4 = 8 bits), the block address (28 bits, assuming 16 bytes blocks and 32-bit addresses), and a command (4 bits). In addition, all replies contain data (16 bytes = 128 bits). For the read-miss request under W-I, two requests (2 × Rr) and two replies containing data (Sw + Rp) are sent. As for the read-exclusive request, we note that a single invalidation is sent which results in three requests (Rxq + Inv + Iack) and one reply (Rxp). Altogether, five requests and three replies are sent. Under AD, four requests (Rr + Mr + DT + MIack) and one reply (Mack) are sent. Thus, the total number of bits sent under W-I to serve a read-miss request and a subsequent read-exclusive request is 704 bits. This number should be compared to 328 bits that are required under AD. To conclude,

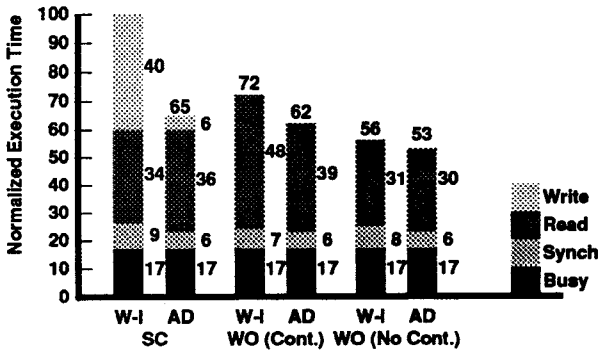


Figure 6: Normalized execution time for AD relative to W-I under SC, assuming sequential consistency (SC) and weak ordering (WO) for the MP3D application.

there is a 53% traffic reduction for each read-miss request to a migratory block under AD. We next study how this reduction affects the overall traffic for the four applications we have run.

In Table 3, we show the traffic reduction data. In MP3D and Water, which contain the largest amount of migratory sharing, traffic is reduced by more than 30%. The traffic reduction in Cholesky, although not as spectacular, is 22% because of a smaller reduction in the number of read-exclusive requests. As we will show, this traffic reduction can have a dramatic impact on performance, even for aggressive network implementations as we assume in this study.

Relaxed consistency models, such as weak ordering [5] and release consistency [6] can hide write stall-time by allowing global requests to overlap each other and local computation. Therefore, one would expect W-I and AD to perform the same under relaxed consistency models. However, as we allow global requests to overlap, the applications will require more bandwidth because the global request rate then increases. Since the adaptive protocol reduces the number of invalidation requests, it is expected to exhibit a lower global access rate. To see this, we measured the execution time for MP3D under weak ordering. We implemented weak ordering by assuming a lockup-free cache [15] that allows an infinite number of global requests to be outstanding as long as synchronizations are respected.

In Figure 6, we show the execution time for MP3D for two consistency model implementations relative to the execution time of W-I under sequential consistency (SC). To the right of SC, we show the execution time under the aggressive weak ordering implementation (WO Cont.). As we would expect, weak ordering manages to hide all write latency for W-I and AD. However, because of a higher global access rate, the read penalty has increased substantially for W-I. As a result, AD performs 16% better under WO. Surprisingly, AD even performs better under SC than does W-I under WO. To confirm that the read penalty increase is because of network contention we ran the same experiments assuming infinite network bandwidth but the same latency. The results are demonstrated by the two rightmost bars in Figure 6 (WO No Cont.). As expected, the performance of W-I and AD are now nearly identical. We also studied the relative performance of AD and W-I under WO for the other applications but did not see any significant differences. The reason for this is that they do not have the same bandwidth requirements as MP3D which can be seen from the larger busy time fractions in Figure 5. Remember that we have assumed a fairly aggressive network implementation (100 MHz synchronous meshes). Therefore, for large system configurations, or networks with less bandwidth such as buses, these effects can show up for applications with less bandwidth requirements than MP3D. A smaller but significant effect from Figure 6 is that the synchronization stall-time is shorter for AD than W-I. The reason for this is due to less contention for critical sections because of global write request reductions.

Table 4: Write penalty reduction (WPR) of W-I by AD and repl. miss-rates (MR) for 64 and 4Kbyte caches.

	MP3D	Cholesky	Water	LU
64 Kbyte — MR	3%	3%	3%	3%
4 Kbyte — MR	7%	18%	9%	21%
64 Kbyte — WPR	86%	67%	94%	3.7%
4 Kbyte — WPR	67%	32%	85%	0.2%

The bottom-line of these experiments is that while WO can hide the write latency it cannot reduce its traffic. The adaptive technique reduces write latency **and** traffic. This is critical to performance for applications with larger bandwidth requirements than the network can sustain.

5.3 Impact of Cache Size on Improvements

In the experiments in the previous sections, we have used a cache size of 64 Kbyte. Since the data sets for our applications are quite small, we end up having virtually no replacement misses as shown in Table 4. To study how the adaptive protocol performs when the replacement miss-rate is higher, we scaled down the cache size to 4 Kbytes.

The effect of replacement misses on migratory sharing is as follows. When a processor has relinquished exclusive access to the block, the block may be replaced from the cache if it is not accessed sufficiently soon by the same processor or invalidated by another processor. As a result, the block is written back to home and the global state becomes Migratory-Uncached under AD and Uncached under W-I. Since the copy is now at home, a subsequent read-exclusive request is performed in at most two network hops, instead of at most three, were the block dirty in

a remote node. As a result, the reduction of write penalty under sequential consistency is expected to be smaller.

In Table 4, we show how much of the write penalty of W-I that is reduced by the adaptive technique for 64 and 4 Kbyte caches. As expected, the performance improvement of AD is now smaller. For example, while AD reduces the write penalty for MP3D by 86% at 64 Kbyte caches, the write penalty reduction drops to 67% at 4 Kbyte. Cholesky shows an even more dramatic change which has to do with the larger increase in replacement miss-rate. Nevertheless, the adaptive protocol still turned out to be effective in detecting migratory sharing.

5.4 Adaptivity to Sharing Behavior Alterations

After a block has been deemed migratory, it will stay in that state unless it starts to be read-only. According to Section 3.4, read-only sharing is detected by the local cache if it receives a migratory read request and the local processor has not written to the block. When this happens, the local cache notifies home by sending a NoMig request (see Figure 1). One could ask whether a block stays migratory for a long time, meaning that NoMig requests are rare. To see this, we measured the fraction of migratory read requests that trigger a NoMig request. We found that these numbers were 0.5%, 0.09%, and 0.01% for MP3D, Cholesky, and Water, respectively. In other words, the migratory sharing that shows up in these applications turns out to be stable. However, we also found that if we disabled this transition, it impacted significantly on the performance which means that this mechanism is needed. Another possibility to make a transition back to write-invalidate depicted in Figure 1 is when home receives read-exclusive requests for blocks nominated as migratory. For all experiments presented in this section, we did not use this heuristic because it did not provide consistent performance improvements.

In summary, the quantitative evaluation has shown that the adaptive protocol can improve performance of a write-invalidate protocol as a result of write-penalty reduction under sequential consistency and read-penalty reduction under relaxed consistency models if the application consumes a lot of bandwidth. We generalize our contributions in the next section.

6 Discussion

We have presented the implementation and evaluation of an adaptive protocol that dynamically can optimize coherence actions due to migratory sharing. Based on the experiments in the previous section, we have found that the adaptive protocol can improve performance of a DASH-like system by reducing the access penalty and network traffic. In this section, we generalize the results to other system organizations and a wider class of applications.

Our implementation is based on a specific cache coherence protocol, in essence the Stanford DASH protocol. However, the adaptive protocol can be built on top of any write-invalidate protocol provided that there is an explicit notion of a home of the coherence state. The detection mechanism relies on the fact that all global read and write requests must interrogate the home directory. In fact, even a COMA architecture that have an explicit home directory for the coherence state, such as the Flat-COMA proposal [16], can use the adaptive protocol. Note also that the protocol is applicable to bus-based systems with snoopy-cache protocols. In such systems a primary concern is to reduce network traffic rather than reducing latency. The adaptive technique is an adequate candidate for such systems.

In our experiments, we considered a rather small system configuration of 16 processors. The implications of our results for larger system configurations are as follows. First, for larger system configurations it will be more difficult to obtain a scalable bandwidth. Secondly, latencies will be larger and thus, the access penalty due to invalidation requests will be higher. The adaptive technique can help performance by reducing both problems.

A limitation of the scope of our results is due to the small number of applications. We found two types of shared data usage that contributed to migratory sharing (i) shared data access protected by locks and (ii) tight read-modify-write operations to shared data. The first type is expected to be common since protected data access is a means to promote correctness. The second type is not so unusual either and happens when a variable is read and modified in the same high-level language statement. It is also interesting to note that migratory sharing is independent of system size. Gupta's and Weber's data of invalidation patterns for 8, 16, and 32 processors [8] support this. They found that the single invalidation numbers for MP3D and Water did not change significantly with system size.

An alternative to the adaptive technique is to use software-controlled, non-binding read-exclusive prefetching [13]. Under this scheme, the programmer/compiler inserts prefetch-instructions so as to get ownership of the block prior to the point when it is accessed. Although this technique can be as effective, it relies on the programmer/compiler to detect the occurrence of read-modify-write operations on shared data which in general can be difficult.

7 Conclusions

The focus of this paper has been to optimize cache coherence actions that arise as a result of migratory objects causing migratory sharing.

We have proposed an adaptive cache coherence protocol that can detect migratory sharing and eliminate all invalidation actions associated with migratory blocks. We

have shown that the mechanisms to support this technique add little to the complexity of coherence mechanisms of directory-based write-invalidate protocols such as the Stanford DASH. Based on a simulation study and four parallel applications, we have found that the adaptive technique can help performance in many important respects. First, because it reduces the number of read-exclusive requests, the write stall-time can be reduced under sequential consistency. Second, even when we go to relaxed consistency models, the read-stall time can be reduced because of contention reduction, especially for applications that require a substantial communication bandwidth. Third, the network traffic was reduced by more than 20% for the studied applications that exhibit migratory sharing, which is as critical for small bus-based as large system configurations.

Because of the limited availability of parallel applications, it is an open question what type of sharing behavior is common and worthwhile to optimize. However, on the premise that migratory objects are common, we feel that the adaptive technique is important because of its simplicity and consistent performance improvements.

Acknowledgments

The authors are deeply indebted to Magnus Karlsson who implemented the DASH simulator. We would also like to thank Kourosh Gharachorloo of Stanford University, Fredrik Dahlgren and Jonas Skeppstedt of Lund University, and the anonymous reviewers for helpful comments. Thanks are also directed to Jeff McDonald, Ed Rothberg, and Jaswinder Pal Singh, who provided us with the parallel applications which made our experiments possible.

This research has been sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK) under the contract number 9001797.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104-114, May 1990.

[2] Kendall Square Research. Kendall Square Research1 (KSR1) Technical Summary. 1992.

[3] J. Boyle et al. Portable Programs for Parallel Processors. Holt, Rinehart, and Winston Inc. 1987.

[4] Mats Brorsson, Fredrik Dahlgren, Håkan Nilsson and Per Stenström. The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors. In *Proceedings of the 26th Annual Simulation Symposium*, to appear, March 1993.

[5] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434-442, 1986.

[6] Kourosh Gharachorloo, Anoop Gupta, John L. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Fourth ASPLOS*, pages 245-257, April 1991.

[7] Kourosh Gharachorloo, Daniel E. Lenoski, James P. Laudon, Philip Gibbons, Anoop Gupta, and John L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15-26, May 1990.

[8] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *Transactions on Computers*, Volume 41, Number 7, pages 794-810, July 1992.

[9] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer Magazine*, pages 44-54, September 1992.

[10] Leslie Lamport. How to make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *Transactions on Computers*. C-28(9), pages 241-248, September 1979.

[11] Daniel E. Lenoski, James P. Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.

[12] Daniel E. Lenoski, James P. Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer Magazine*, pages 63-79, March 1992.

[13] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 2(4), pages 87-106, June 1991.

[14] Jaswinder P. Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1). pages 5-44, March 1992.

[15] Per Stenström, Fredrik Dahlgren, and Lars Lundberg. A Lockup-free Multiprocessor Cache Design. In Proceedings of *1991 International Conference on Parallel Processing*, Vol. I, pages 246-250, August 1991.

[16] Per Stenström, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80-91, May 1992.