An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing

Liqun Cheng and John B. Carter
University of Utah
{*legion, retrac*}*@cs.utah.edu*

Donglai Dai
Silicon Graphics, Inc.
dai@sgi.com

Abstract

Shared memory multiprocessors play an increasingly important role in enterprise and scientific computing facilities. Remote misses limit the performance of shared memory applications , and their significance is growing as network latency increases relative to processor speeds.

This paper proposes two mechanisms that improve shared memory performance by eliminating remote misses and/or reducing the amount of communication required to maintain coherence. We focus on improving the performance of applications that exhibit producer-consumer *sharing. We first present a simple hardware mechanism for detecting producer-consumer sharing. We then describe a* directory delegation *mechanism whereby the "home node" of a cache line can be delegated to a producer node, thereby converting 3-hop coherence operations into 2-hop operations. We then extend the delegation mechanism to support* speculative updates *for data accessed in a producer-consumer pattern, which can convert 2-hop misses into local misses, thereby eliminating the remote memory latency. Both mechanisms can be implemented without changes to the processor.*

We evaluate our directory delegation and speculative update mechanisms on seven benchmark programs that exhibit producer-consumer sharing using a cycle-accurate execution-driven simulator of a future 16-node SGI multiprocessor. We find that the mechanisms proposed in this paper reduce the average remote miss rate by 40%, reduce network traffic by 15%, and improve performance by 21%. Finally, we use Murphi to verify that each mechanism is error-free and does not violate sequential consistency.

1 Introduction

Most enterprise servers and many of the Top500 supercomputers are shared memory multiprocessors [1]. Remote misses have a significant impact on shared memory performance, and their significance is growing as network hop latency increases as measured in processor cycles [9, 13, 14, 16, 26, 33]. The performance impact of remote memory accesses can be mitigated in a number of ways, including higher performance interconnects, sophisticated processor-level latency hiding techniques, and more effective caching and coherence mechanisms. This paper focuses on the latter.

Previous research has demonstrated the value of adaptive protocols that identify and optimize for migratory sharing [10, 32]. In this paper, we present a novel adaptive mechanism that identifies and optimizes for producer-consumer sharing. An important feature of the mechanisms described herein is that they *require no changes to the processor core or system bus interface* and can be implemented entirely within an external directory controller. In addition, *our optimizations are sequentially consistent.* Further, *we identify producer-consumer sharing with a very simple mechanism* that does not require large history tables or similar expensive structures.

Typical cc-NUMA systems maintain coherence using directory-based distributed write-invalidate protocols. Each cache line of data has a "home node" that tracks the global coherence state of the cache line, e.g., which nodes have a cached copy of the data, via a directory structure [23]. Before a processor can modify a cache line of data, it must invalidate all remote copies and be designated as its *owner*. Thus, only the first write in a sequence of writes to a cache line with no intervening read operations from other nodes causes global actions. This behavior leads to good performance for situations where data is used exclusively by a single processor or mostly read-shared.

However, write invalidate protocols are inefficient for some sharing patterns. For example, consider producer-consumer sharing, where a subset of threads access a shared data item, one of which modifies the data while the others read each modification. Figure 1 illustrates the communication induced by producer-consumer sharing involving one producer (P) and two consumer nodes (C1 and C2), where data is initially cached in read-only mode on each node. Solid lines represent request messages, dashed lines represent intervention messages (e.g., an invalidation message), and dotted lines represent intervention replies.

Unless the producer is located on the home node, a conventional directory-based write-invalidate protocol

1-4244-0805-9/07/$25.00 ©2007 IEEE

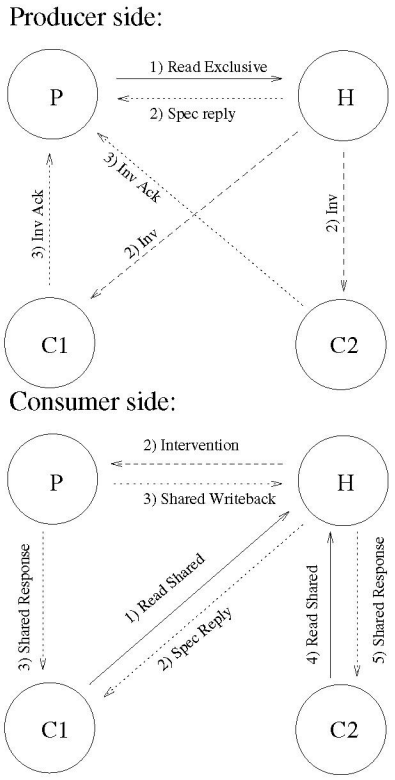

Figure 1. Traditional Producer-Consumer

requires at least three network latencies whenever a producer wishes to modify data: (1) a message from the producer to the home node requesting exclusive access, (2) a message from the home node to the consumer(s) requesting that they invalidate their copy of the data, and (3) acknowledgments from the home node and consumers to the producer. The top half of Figure 1 illustrates this scenario. Similarly, when the first consumer accesses the data after its copy has been invalidated, it incurs a 3-hop miss unless it or the producer is located on the home node: (1) a message from the consumer to the home node requesting a read-only copy of the data, (2) a message from the home node to the producer requesting that the producer downgrade its copy to SHARED mode and write back the new contents of the cache line, and (3) a message from the producer to the consumer providing the new contents of the data and the right to read. The bottom of Figure 1 illustrates this scenario.

One performance problem illustrated in Figure 1 is the third network hop required when the producer does not reside on the home node. To address this problem, we propose a novel *directory delegation* mechanism whereby the "home node" of a particular cache line of data can be delegated to another node. During the period in which the directory ownership is delegated, the home node forwards requests for the cache line to the delegated home node. Other nodes that learn of the delegation can send requests directly to the delegated node, bypassing the home node as long as the delegation persists. When used appropriately, directory delegation can convert a number of 3-hop coherence operations into 2-hop operations.

To improve producer-consumer sharing performance, we extend the delegation mechanism to enable the producer (delegated home node) to speculatively forward newly written data to the nodes that it believes are likely to consume it in the near future. Our update mechanism is a performance optimization on top of a conventional write invalidate protocol, analogous to prefetching or last write prediction [21], and thus does not affect the (sequential) consistency semantics of the underlying coherence protocol. When the producer correctly predicts when and where to send updates, 2-hop misses become local misses, effectively eliminating the impact of remote memory latency.

Although update protocols have the potential to eliminate remote misses by pre-pushing data to where it will soon be consumed, they have a tendency to generate excessive amounts of coherence traffic [7, 15]. These extra coherence messages are typically due to sending updates to nodes that no longer are consuming the data. This extra coherence traffic can lead to serious performance problems by consuming precious interconnect bandwidth and memory controller occupancy. To mitigate this problem, we only perform speculative updates when the producer has been delegated control of a cache line and a stable producer-consumer sharing pattern has been observed. For the benchmarks we examine, the speculative push mechanism generates less network traffic than even a tuned write invalidate protocol.

A goal of our work is to design mechanisms that can be implemented in near future systems. We concentrate on mechanisms that require only modest hardware overhead, require no changes to the processor, do not add significant pressure to the interconnect, and do not significantly increase directory controller occupancy. We do not consider designs that require large history tables, assume the ability to "push" data into processor caches, perform updates that are not highly likely to be consumed, or make other assumptions that make it difficult to apply our ideas to commercial systems. In Section 5 we discuss ways that our work can be extended by relaxing some of these restrictions.

Using a cycle-accurate execution-driven simulator of a future-generation 16-node SGI multiprocessor, we examine the performance of seven benchmarks that exhibit varying degrees of producer-consumer sharing. We find that our proposed mechanisms eliminate a significant fraction of remote misses (40%) and interconnect traffic (15%), leading to a mean performance improvement of 21%. The reduction in both coherence traffic

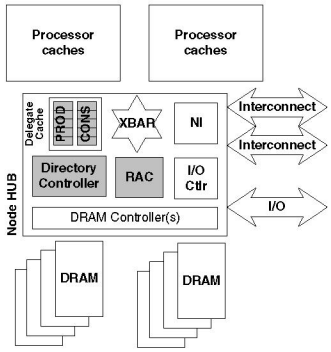

Figure 2. Modeled node architecture (modified/new components are highlighted)

and remote misses plays a significant role in the observed performance improvements. The hardware overhead required for the optimizations described above is less than the equivalent of 40KB of SRAM per node, including the area required for the delegate cache, remote access cache, and directory cache extensions used to detect producer-consumer sharing.

2 Protocol Implementation

In this section we describe the design of our various novel coherence mechanisms (producer-consumer sharing predictor, directory delegation mechanism, and speculative update mechanism). We begin with a brief discussion of remote access caches and their value in our work (Section 2.1). We then describe the simple hardware mechanism that we employ to identify producer-consumer sharing patterns. Then, in Section 2.3 we describe the protocol extensions and hardware needed to support directory delegation. In Section 2.4 we describe the protocol extensions and hardware required to implement our speculative update mechanism. Finally, in Section 2.5, we describe how we verified the correctness of our protocol extensions using the Murphi model checker.

2.1 Remote Access Cache

Traditional CC-NUMA machines can only store remote data in their processor caches. Remote access caches (RACs) eliminate unnecessary remote misses induced by the small size and associativity of processor caches by acting as victim caches for remote data [23]. However, the large caches in modern processors have largely eliminated remote conflict and capacity misses, so modern shared memory multiprocessors do not include RACs.

We propose to augment each node's "hub" with a RAC for three reasons. First, the RAC can be used as a traditional victim cache for remote data. Second, the RAC gives us a location into which we can *push* data at a remote node; researchers often assume the ability to push data into processor caches, but this capability is not supported by modern processors. Third, we employ a portion of the RAC as a surrogate "main memory" for cache lines that have been delegated to the local node. In particular, for each cache line delegated to the local node, we pin the corresponding cache line in the local RAC.

2.2 Sharing Pattern Detection

To exploit delegation and speculative updates, we first must identify cache lines that exhibit a stable producer-consumer sharing pattern. In this section we describe the producer-consumer pattern detector that we employ.

We define producer-consumer sharing as a repetitive pattern wherein one processor updates a block (producer) and then an arbitrary number of processors read the block (consumer(s)), which corresponds to the following regular expression:

$$...(W_i)(R_{\forall j:j\neq i})^+(W_i)(R_{\forall k:k\neq i})^+... \quad (1)$$

In the expression above, R_i and W_i represent read and write operations by processor i, respectively.

Coherence predictors can be classified into two categories: instruction-based [18] and address-based [29]. Instruction-based mechanisms require less hardware overhead, but require tight integration into the processor core. Since one of our goals is to require no changes to the processor core, we focus on address-based predictors. A problem with address-based predictors implemented outside of the core is they can only observe addresses of accesses that miss in the processor caches. Also, they typically require extensive storage to track access histories, e.g., Lai et al. [20] add one history entry per memory block to trace sharing patterns, which is roughly a 10% storage overhead.

To address these problems, we extend each node's directory controller to track block access histories. Since the directory controller coordinates all accesses to memory blocks homed by that node, it has access to the global access history of each block that it manages. To minimize space overhead, we only track the access histories of blocks whose directory entries reside in the directory cache, not all blocks homed by a node. Typically, a directory cache only contains a small number of entries, e.g., 8k entries on SGI Altix systems, which corresponds to only a fraction of the total memory homed on a particular node. However, the blocks with entries in the directory cache are the most recently shared blocks homed on that node. We found that tracking only their access histories detects the majority of the available

producer-consumer sharing patterns, which corroborates results reported by Martin *et al.* [24].

To detect producer-consumer sharing, we augment each directory cache entry with three fields: *last writer*, *reader count*, and a *write-repeat counter*. The last writer field (4 bits) tracks the last node to perform a write operation. The reader count (2 bits, saturating) remembers the number of read requests from unique nodes since the last write operation. The write-repeat counter (2 bits, saturating) is incremented each time two consecutive write operations are performed by the same node with at least one intervening read. Our detector marks a memory block as being producer-consumer whenever the write-repeat counter saturates. These extra 8 bits, which increase the size of each directory cache entry by 25%, are not saved if the directory entry is flushed from the directory cache.

The detector logic we employ is very simple, which limits its space overhead but results in a conservative predictor that misses some opportunities for optimization, e.g., when more than one node writes to a cache line. An important area of future work is to experiment with more sophisticated producer-consumer sharing detectors. However, as discussed in Section 3, even this simple detector identifies ample opportunities for optimization.

2.3 Directory Delegation

Once the predictor identifies a stable producer-consumer sharing pattern, we delegate directory management to the producer node, thereby converting 3-hop misses into 2-hop misses. To support delegation, we augment the directory controller on each node with a *delegate cache*. The delegate cache consists of two tables: a *producer table* that tracks the directory state of cache lines delegated to the local node and a *consumer table* that tracks the delegated home node of cache lines being accessed by the local node. The producer table is used to implement delegated directory operations. Its entries include the directory information normally tracked by the home node (DirEntry), a valid bit, a tag, and an age field. The number of cache lines that can be delegated to a particular node at a time is limited by the size of the producer table. The consumer table allows consumers to send requests directly to the delegated (producer) node, bypassing the default home node. Its entries consist of a valid bit, a tag, and the identity of the new delegated home node for the corresponding cache line. The number of cache lines that a node will recognize as having a new (remote) home node is limited by the size of the consumer table. The format of delegate cache entries are shown in Figure 3.

When a node needs to send a message to a cache line's home, e.g., to acquire a copy or request ownership, it first checks the local delegate cache. If it finds a corresponding entry in the producer table, it handles the request locally. If it finds an entry in the consumer table, it forwards the request to the delegated home node recorded in the entry. Otherwise, it sends the request to the default home node.

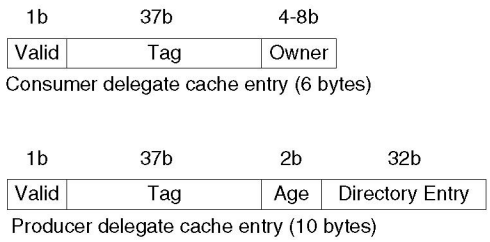

Figure 3. Format of delegate cache entries

The following subsections describe how we initiate directory delegation (Section 2.3.1), how coherence requests are forwarded during delegation (Section 2.3.2), and how we un-delegate nodes (Section 2.3.3).

2.3.1 Delegation

After the home detects a stable producer-consumer sharing pattern, it marks the producer as the new owner of the cache line, changes the directory state to DELE, and sends the producer a DELEGATE message that includes the current directory information. Upon receiving this message, the producer adds an entry to its producer delegate table, treats the message as an exclusive reply message, and pins the corresponding RAC entry so that there is a place to put the data should it be flushed from the processor caches.

2.3.2 Request Forwarding

While the cache line is in the DELE state on the original home node, coherence messages sent to the original home node are forwarded to the delegated home node. The original home node also replies to the requester to notify it that the producer is acting as the home node until further notice, in response to which the requesting node adds an entry in its consumer table in the delegate cache. Subsequent coherence requests from this node are sent directly to the delegated home node until the line is undelegated.

Entries in the consumer table are *hints* as to which node is acting as the home node for a particular cache line, so it can be managed with any replacement policy. If a consumer table entry becomes stale or is evicted, the consumer can still locate the acting home node, but will require extra messages to do so. In our design, the consumer table is 4-way set associative and uses random replacement.

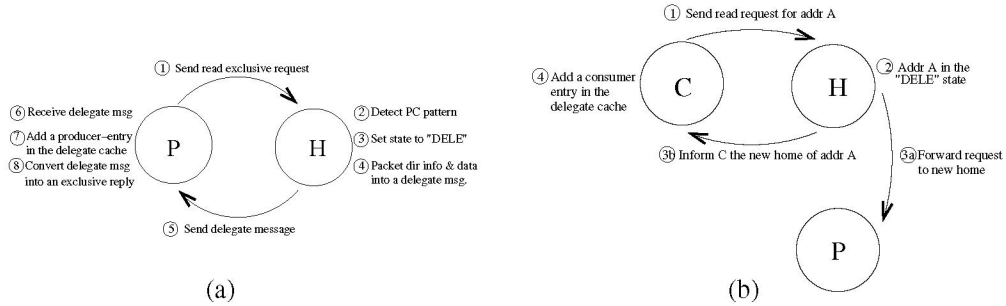

Figure 4. Operations involved in (a) directory delegation and (b) request forwarding

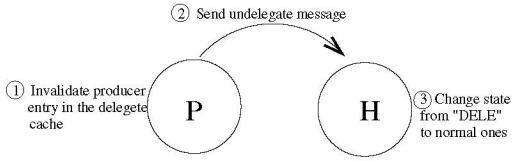

Figure 5. Directory undelegation

2.3.3 Undelegation

In our current design, a node undelegates control over a cache line for three reasons:

1. the delegated home node runs out of space in its producer table and needs to replace an entry,
2. the delegated home node flushes the corresponding cache line from its local caches, or
3. another node requests an exclusive copy.

To undelegate a cache line, the producer node invalidates the corresponding entry in its producer table and sends an UNDELE message to the original home node, including the current directory state (DirEntry) and cache block contents (if dirty). When the original home node receives the UNDELE message, it changes the local state of the cache line from DELE to UNOWNED or SHARED, depending on the contents of DirEntry. If the reason for undelegation is a request for exclusive ownership by another node, the UNDELE message includes the identity of this node and the original home node can handle the request.

2.3.4 Discussion

There are a number of race conditions and corner cases that need to be handled carefully to make delegation, forwarding, and undelegation work correctly. For example, it is possible for a request by the producer to arrive at the original home node while it is in the process of delegating responsibility to the producer. Or, the producer may receive a coherence request for a line for which it has undelegated responsibility and for which it no longer has any information in its producer table. To handle these kinds of races, we employ the mechanism used by SGI to simplify its coherence protocol implementations: NACK and retry. In the first case the original home node NACKs the request from the producer, causing it to retry the operation. When the producer retries the operation it will most likely find that it has become the acting home node, otherwise its requests will be NACKed until the delegation operation succeeds. Similarly, if the producer receives a coherence operation for a line that it undelegated, it NACKs and indicates that it is no longer the acting home node, which causes the requesting node to remove the corresponding entry from its consumer table and re-send the request to the default home node. In Section 2.5, we describe how we formally verify the correctness of our delegation, forwarding, and undelegation protocols.

2.4 Speculative updates

On conventional cc-NUMA systems, applications with significant producer-consumer sharing suffer from frequent remote read misses induced by the invalidations used to ensure consistency. To reduce or eliminate these remote read misses, we extend our directory delegation mechanism to support selective updates. To ensure sequential consistency and simplify verification, we implement our update mechanism as an optimization applied to a conventional write invalidate protocol, rather than replacing the base coherence protocol with an update protocol. Reads and writes continue to be handled via a conventional directory-based write invalidate protocol. However, after a cache line has been identified as exhibiting producer-consumer sharing and been delegated, producers send speculative update messages to the identified consumers shortly after each invalidation. In essence, we support a producer-driven "speculative push", analogous to a consumer-driven prefetch mechanism. These pushes are a performance optimization and do not impact correctness or violate sequential consistency.

To support speculative updates, we must overcome two challenges. The first is determining when to send updates and what data to send *without modifying the processor*. The second is limiting the use of updates to situations where we have high confidence that the updates will be consumed. We address these challenges in the following subsections.

2.4.1 Delayed intervention

Modern processors directly support only invalidate protocols. Before performing a write, processor requests exclusive access to a cache line. After becoming the exclusive owner, a processor is free to modify it as many times as it likes and can refrain from flushing those changes until another processor attempts to access the cache line.

To generate updates without modifying the processor, we employ a *delayed intervention* mechanism. When the producer wishes to modify a cache line, it requests exclusive access to the line. As in a normal write invalidate protocol, we invalidate all other nodes that have a shared copy. After the producer's hub grants the producer exclusive access to the data, it delays for a short period and then sends an intervention request to the processor, requesting that it downgrade the cache line to SHARED mode. In response to this intervention, the processor flushes the dirty contents of the cache line. The producer's hub writes the flushed data into the local RAC and then sends update messages to the predicted consumers. The mechanism via which we predict the consumer set is described below.

Our delayed intervention mechanism could be thought of as an extremely simple last write predictor, analogous to Lai et al.'s last touch predictor [21]. To avoid modifying the processor, we simply wait a short time and then predict that the current write burst has completed, rather than employing large tables to track past access histories. Lai et al. used last touch prediction to dynamically self-invalidate cache lines, whereas we employ last write prediction to dynamically downgrade from EXCL to SHARED mode and generate speculative updates. By self-invalidating, Lai et al. convert read misses from 3-hop misses to 2-hop misses, whereas we use updates to convert 3-hop misses into 0-hop (local) misses. The downside of an overly aggressive downgrade decision has less impact in our design, since the producer retains a SHARED copy of the block. For these reasons, a less accurate predictor suffices to achieve good performance improvements.

It is important that delayed intervention not occur too soon or too long after data is supplied to the processor. If too short of an interval is used, the processor may not have completed the current write burst and will suffer another write miss. Overly long intervals result in the data not arriving at consumers in time for them to avoid suffering read misses. For simplicity, we use a fixed (configurable) intervention delay of 50 processor cycles. In Section 3.3 we show that performance is largely insensitive to delay intervals between 5 and 5000 cycles. Developing a more adaptive intervention mechanism is part of our future plans.

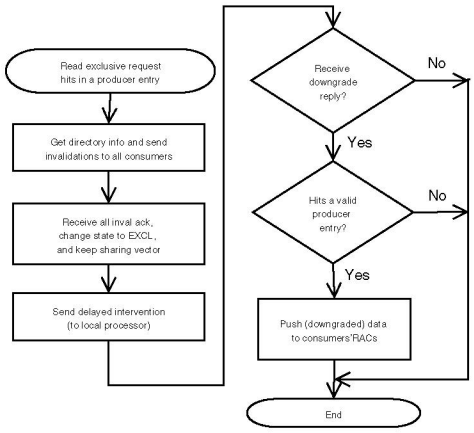

Figure 6. Flow of speculative updates

2.4.2 Selective updates

To achieve good performance, we must accurately predict which nodes will read an update before the cache line is modified again. An overly aggressive mechanism will generate many useless updates, while an overly conservative mechanism will not eliminate avoidable remote read misses. To limit the number of unnecessary updates, we only send updates for cache lines that exhibit producer-consumer behavior and only send updates to nodes that were part of the sharing vector, i.e., the nodes that consumed the last update. Due to the way producer-consumer sharing works, these are the nodes most likely to consume the newly written data. We track these nodes as follows.

In a traditional invalidated-based protocol, when the state changes from SHARED to EXCL, the sharing vector in the corresponding directory is replaced by the NodeID of the exclusive owner. To track the most recent consumer set, we add an ownerID field to the directory entry and use the old sharing vector to track the nodes to send updates, only overwriting it when a new read request is received.

2.4.3 Summary of Control Flow

Figure 6 shows the control flow of our speculative update operation. The producer node starts with a shared copy of the cache line that has already been delegated. When the producer writes the data, the local coherence engine loads the directory information from the producer entry in to the local delegate cache. After invalidating all shared copies, the producer's hub changes the state to EXCL, without overwriting the sharing vector, and gives the requesting processor exclusive access. Several cycles later, the hub issues a delayed intervention on the local system bus, which causes the producer processor to downgrade the cache line and write back its new contents. After receiving the new data, the hub sends an update to each node listed in the sharing vector. Upon receipt of an update, a consumer places the incom-

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
System bus	16B CPU to system, 8B system to CPU
	max 16 outstanding L2C misses, 1GHZ
DRAM	4 16-byte-data DDR channels
Hub clock	500 MHz
DRAM	200 processor cycles latency
Network	100 processor cycles latency per hop

Table 1. System configuration.

ing data in the local RAC. If the consumer processor has already requested the data, the update message is treated as the response.

2.5 Verification

We formally verified that the mechanisms described above do not violate sequential consistency, introduce deadlocks or livelocks, or have other race conditions or corner cases that violate correctness. We applied the standard method for debugging cache coherence protocols: we built a formal model of our protocols and performed an exhaustive reachability analysis of the model for a small configuration size [28, 25, 8] using explicit-state model checking with the Murphi [11] model checker. We extended the DASH protocol model provided as part of the Murphi release, ran the resulting model through Murphi, and found that none of the invariants provided in the DASH model were violated by our changes. Moreover, we applied invariant checking to our simulator to bridge the gap between the abstract model and the simulated implementation and again found that no invariants are violated. More specifically, we tested both Murphi's "single writer exists" and "consistency within the directory" invariants at the completion of each transaction that incurs a L2 miss.

3 Evaluation

3.1 Simulator Environment

We use a cycle-accurate execution-driven simulator, UVSIM, in our performance study. UVSIM models a hypothetical future-generation SGI Altix architecture that we are investigating along with researchers from SGI. Table 1 summarizes the major parameters of our simulated system. The simulated interconnect is based on SGI's NUMALink-4, which uses a fat-tree structure with eight children on each non-leaf router. The minimum-sized network packet is 32 bytes and we model a network hop latency of 50nsecs (100 CPU cycles). We do not model contention within the routers, but do model hub port contention.

Application	Problem size
Barnes	16384 nodes, 123 seed
Ocean	258*258 array, 1e-7 error tolerance
Em3D	38400 nodes, degree 5, 15% remote
LU	16*16*16 nodes, 50 testes
CG	1400 nodes, 15 iteration
MG	32*32*32 nodes, 4 steps
Appbt	16*16*16 nodes, 60 timesteps

Table 2. Applications and data sets

Application	Number of Consumers (%)				
	1	2	3	4	4+
Barnes	13.9	6.8	9.4	8.1	61.7
Ocean	97.7	1.8	0.5	0	0
Em3D	67.8	32.2	0	0	0
LU	99.4	0	0	0.4	0.1
CG	0.1	0.2	0	0	99.7
MG	78.3	11.4	3.7	2.6	3.9
Appbt	0	0.3	6.7	1.4	91.6
Average	51.0	7.5	2.9	1.8	36.7

Table 3. Number of consumers in the producer-consumer sharing patterns

3.2 Results

We model a 16-processor system. Table 2 presents the input data sets of the applications we use in this study. We consider a mix of scientific applications that exhibit varying degrees of producer-consumer sharing. Barnes and Ocean (contig) are from the SPLASH-2 benchmark suite [35]; EM3D is a shared-memory implementation of the Split-C benchmark; LU, CG, MG and Appbt are NAS Parallel Benchmarks [12]. We use Omni's OpenMP versions of the NPBs [2].

All results reported in this paper are for the parallel phases of these applications. Data placement is done by SGI's first-touch policy, which tends to be very effective in allocating data to processors that use them.

Figure 7 presents the execution time speedup, remote miss reduction, and network message reduction for each application for a range of machine configurations. All results are scaled relative to the baseline system runs. For each application, we also show the results for a system with a 32K RAC and no delegation or update mechanisms, a system with 32-entry delegate tables and a 32K RAC, and a system with 1K-entry delegate tables and a 1M RAC. Other than the baseline and RAC-only systems, the results include both directory delegation and selective updates. We omit results for delegation-only, as we found that the benefit of turning 3-hop misses into 2-hop misses roughly balanced out the overhead of delegation, which resulted in performance within 1% of the baseline system for most applications.

Barnes simulates the interaction of a system of bodies in three dimensions using the Barnes-Hut hierarchi-

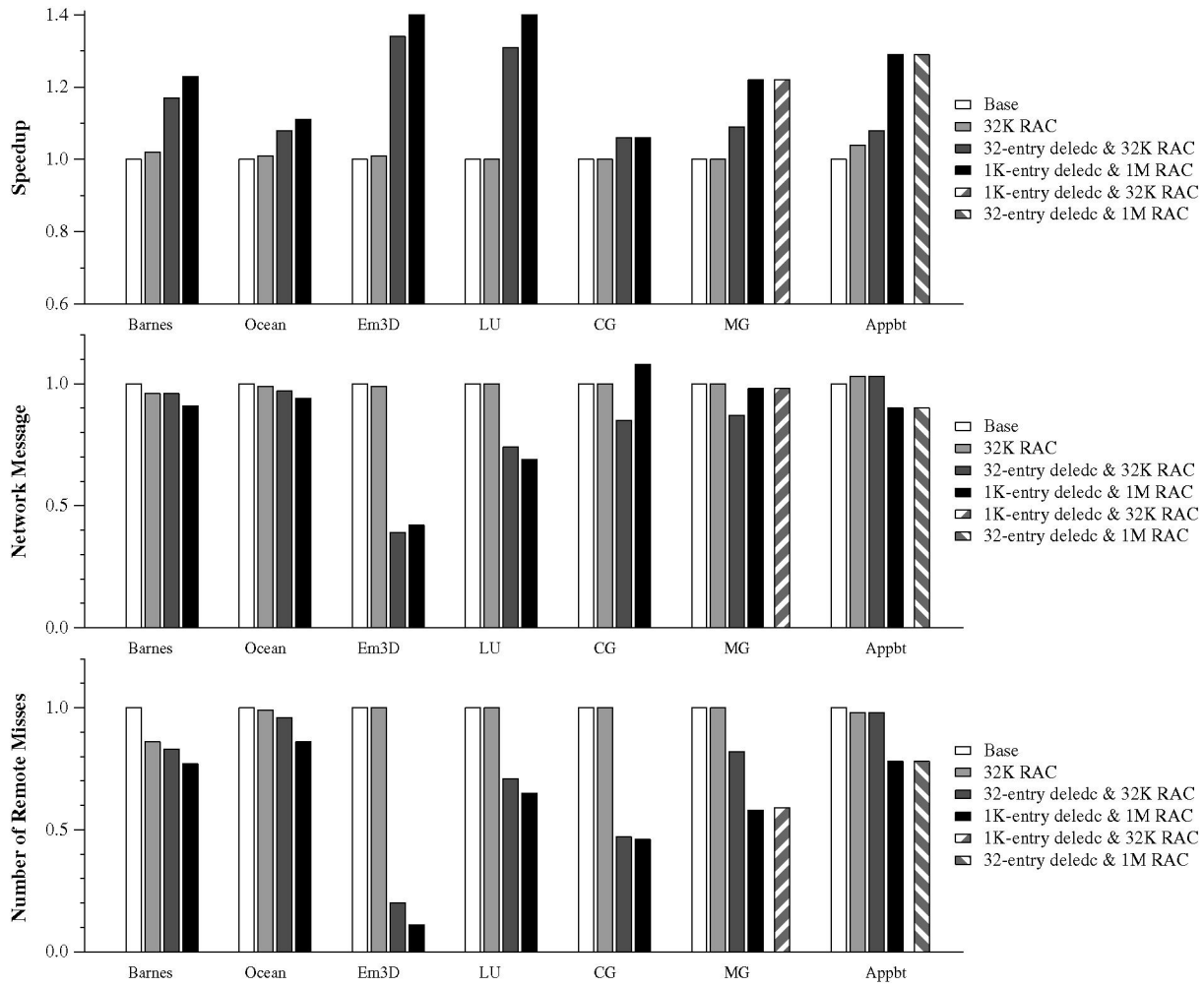

Figure 7. Application speedup, network messages, and remote misses

cal N-body method. The main data structure is an octree with leaves containing information on each body, and internal nodes representing space cells. During each iteration, processors traverse the octree to calculate forces between the bodies and rebuild the octree to reflect the movement of bodies. Although the communication patterns are dependent on the particle distribution, which changes with time, barnes exhibits stable producer-consumer sharing patterns during each phase. The octree data structure inherently results in there being a significant number of consumers per producer, as shown in Table 3. Therefore, the benefit of selective updates within each phase are substantial. Even a small delegate cache and RAC (32-entries and 32K respectively) eliminates roughly 20% of the baseline version's remote misses, which leads to a 17% performance improvement. This performance improvement grows to 23% for the larger delegate cache/RAC configuration.

Ocean models large-scale ocean movements based on eddy and boundary currents. Processors communicate with their immediate neighbors, so nodes along processor allocation boundaries exhibit single producer single consumer sharing. The small number of consumers per producer limits the potential benefits of our speculative update mechanism, as illustrated by the modest remote miss reduction and performance benefits (8% for the small RAC/delegate cache configuration and 11% for the larger one).

Em3d models the propagation of electromagnetic waves through objects in three dimensions. It includes two configuration parameters that govern the extent of producer-consumer sharing: *distribution span* indicates how many consumers each producer will have while *remote links* controls the probability that the producer and consumer are on the different nodes. We use a *distribution span* of 5 and *remote links* probability of 15%.

Selective updates improve performance by 33-40% due largely to a dramatic reduction in coherence message traffic (60%) and an even larger reduction in remote misses (80-90%). The benefits of eliminating remote misses are obvious, but there are several other phenomenon at work that help eliminate network traf-

fic. First, delegating the home directory to the producer node converts 3-hop misses in to 2-hop misses. A less obvious source of traffic reduction comes from eliminating NACKs induced when barrier synchronization is implemented on top of a write invalidate protocol. After a barrier is crossed, a large number of nodes often attempt to read the same invalidated cache line simultaneously, which causes congestion at the data's home directory. To avoid queueing and potential deadlocks, the directory NACKs requests for a particular cache line if it is BUSY performing another operation on that line. In em3d, NACK messages caused by this "reload flurry" phenomenon represent a nontrivial percentage of network traffic and are largely removed by speculative updates.

LU solves a finite difference discretization of the 3D compressible Navier-Stokes equations through a block-lower block-upper approximate factorization of the original difference scheme. The LU factored form is solved using successive over-relaxation (SOR). A 2D partitioning of the grid onto processors divides the grid repeatedly along the first two dimensions, alternately x and then y, which results in vertical columns of data being assigned to individual processors. Boundary data exhibits stable producer-consumer sharing. Again, even a small delegate cache and RAC are able to achieve good performance improvements (31% speedup, 26% traffic reduction, and 30% remote miss reduction), while a larger configuration achieves a 40% speedup, 30% traffic reduction, and 35% remote miss reduction.

CG uses the conjugate gradient method to compute an approximation of the smallest eigenvalue of a large sparse symmetric positive definite matrix. Multiple issues limit the speedup of CG to 6%. First, CG exhibits producer-consumer sharing only during some phases. Second, the sparse matrix representation used in CG exhibits a high degree of false sharing. Our simple cache line-grained producer-consumer pattern detector avoids designating such cache lines as good candidates for selective updates, which limits the potential performance benefits. Third, and most important, remote misses are not a major performance bottleneck, so even removing roughly 60% of them does not improve performance dramatically.

MG is a simplified multigrid kernel that solves four iterations of a V-cycle multigrid algorithm to obtain an approximate solution to a discrete Poisson equation. The V-cycle starts with the finest grid, proceeds through successive levels to the coarsest grid, and then walks back up to the finest grid. At the finest grid size, boundary data exhibits producer-consumer sharing. At coarse grid sizes, two pieces of dependent data are most likely on different processors, so more data exhibits producer consumer sharing. In fact, a 32-entry delegate cache is too small to hold all cache blocks identified as exhibiting producer-consumer sharing, which limits the number of remote misses that can be removed to 20% and the performance improvement to 9%. Increasing the delegate cache size to include 1K-entry tables increases the performance benefit to 22%, even if we leave the RAC size at 32KB. Note that the larger configuration results in almost the same network traffic as the baseline system, largely to due being overly aggressive in sending out updates. As expected, eliminating remote misses has a larger impact on performance than eliminating network traffic, so the larger configuration achieves better performance despite being less effective at eliminating network traffic.

Appbt is a three dimensional stencil code in which a cube is divided into subcubes that are assigned to separate processors. Gaussian elimination is then performed along all three dimensions. Like MG, Appbt exhibits significant producer-consumer sharing along subcube boundaries. Like MG, the small RAC/delegate cache is able to capture only a fraction of the performance benefit of the large configuration, 8% compared to the 24% speedup achieved by the large configuration. In this case, the small RAC is the performance bottleneck, and increasing the RAC size to 1MB while retaining the 32-entry delegate cache tables achieves virtually all of the benefit of the large configuration.

Overall, the geometric mean speedup of our speculative update mechanism across the seven benchmark programs using the small RAC/delegate cache is 13%, while network traffic is reduced by an arithmetic mean of 17% and 29% of remote misses are eliminated. If we increase the size of the delegate cache tables to 1K entries and the RAC size to 1MB, mean speedup increases to 21%, with a 15% reduction in message traffic and 40% reduction in remote misses.

In general, the primary benefit of speculative updates comes from removing remote misses, especially for em3d and MG. However, in some cases eliminating a significant number of remote misses does not translate directly into large runtime improvements.

3.3 Sensitivity Analysis

In this section, we evaluate how sensitive the results are to various design choices and system parameters.

3.3.1 Delegation/updates versus larger caches

To support 32-entry delegate tables and a 32KB RAC requires roughly 40KB of extra SRAM per node, plus a small amount of control logic and wire area. This estimate is derived as follows. A 32-entry delegate table requires 320 bytes. Extending the directory cache to support the sharing pattern predictor adds 8 bits to each directory entry (a 4-bit last writer field, a 2-bit reader counter, and a 2-bit write-repeat counter), which for a

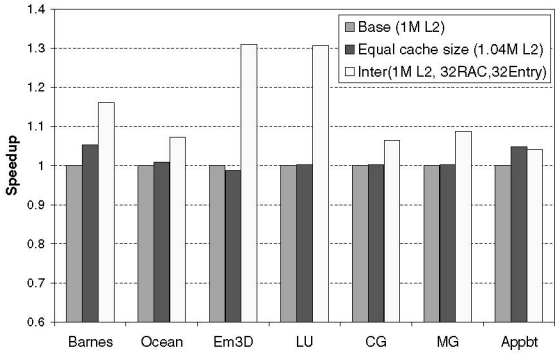

Figure 8. Equal storage area comparison

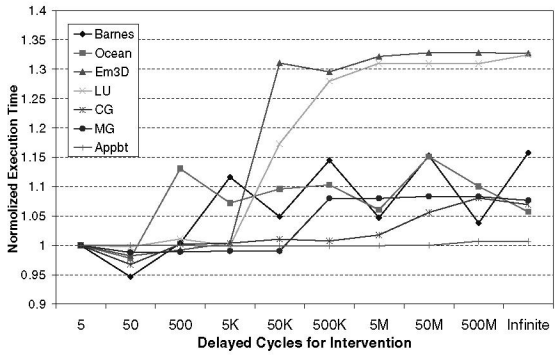

Figure 9. Sensitivity to intervention delay

8192-entry (32KB) directory cache represents an extra 8KB of storage. We do not save these extra bits when a directory entry is flushed, so there is no extra main memory overhead.

In Figure 8 we present the performance of a system with 1MB of L2 cache and no extensions, a system with 1MB of L2 cache extended with a 32-entry delegate cache and a 32KB RAC, and an equal silicon area system with 1.04MB of L2 cache and no extensions. Note that a 1.04MB cache involves adding silicon to the processor die, but we include it to compare the value of building "smarter" versus "larger" caches.

For most benchmarks adding a 32-entry delegate cache and a 32KB RAC yields significantly better performance than simply building a larger L2 cache. The exception is Appbt. Recall that Appbt requires a large RAC to hold all producer-consumer data; a small RAC such as the one modeled here suffers excessive RAC misses, which limits the performance improvement of delegation and updates.

3.3.2 Sensitivity to intervention delay interval

Figure 9 presents the execution time of each application as we vary the delay interval from 5 to 500M cycles normalized to the performance with a 5-cycle delay. A delay interval of 5 cycles results in 1%-5% worse performance than a 50-cycle delay interval, because some write bursts last longer than 5 cycles, which causes extra write misses and updates. As we increase the delay interval beyond 50 cycles, performance degrades at different rates for different applications. Some applications can tolerate a higher delay interval, e.g., MG performance does not deteriorate until the delay interval exceeds 50K cycles, because the average latency between producer writes and consumer reads is large. A delay interval of 50 cycles works well for all of the benchmarks because it is long enough to capture most write bursts, but short enough to ensure that updates arrive early enough to eliminate consumer read misses.

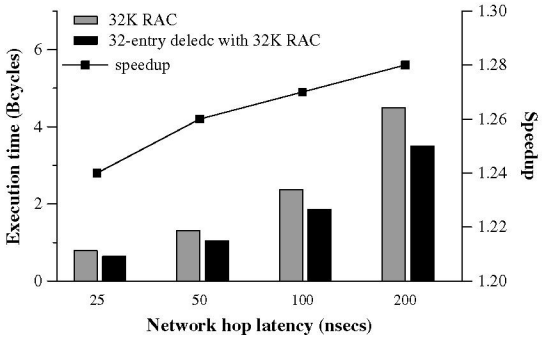

Figure 10. Sensitivity to hop latency (Appbt)

The performance of Ocean and Barnes varies inconsistently (up and down) as we vary the delay interval. This phenomenon is caused by a non-obvious interaction between the choice of delay interval and the timing of synchronization in these applications. When the choice of delay interval happens to cause updates to occur at the same time that the application is synchronizing between threads, e.g., as occurs for both Ocean and Barnes with a delay interval of 5M cycles, performance suffers.

3.3.3 Sensitivity to Network Latency

To investigate the extent to which remote miss latency impacts performance, we vary the network hop latencies from 25nsecs to 200nsecs. Figure 10 plots the execution time (left y-axis) and speedup (right y-axis) for Appbt, whose performance is representative. We consider only the baseline CC-NUMA system and a system enhanced with a 32K RAC and 32-entry delegate cache tables. Every time network hop latency doubles, execution time nearly doubles. Therefore, the value of the mechanisms proposed herein increases as average remote miss latencies increase, albeit only gradually (increasing from a 24% speedup to 28% as we increase hop latency from 25nsecs to 200nsecs).

3.3.4 RAC and Delegate Cache

For most benchmarks, a small delegate cache and RAC (32-entry and 32KB) result in significant performance benefits. Two exceptions are MG and Appbt, which are limited by the size of the delegate cache and RAC, respectively. Figure 11 shows MG's sensitivity to the

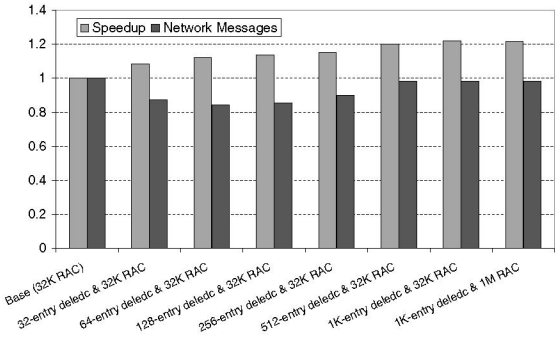

Figure 11. Sensitivity to delegate cache sz (MG)

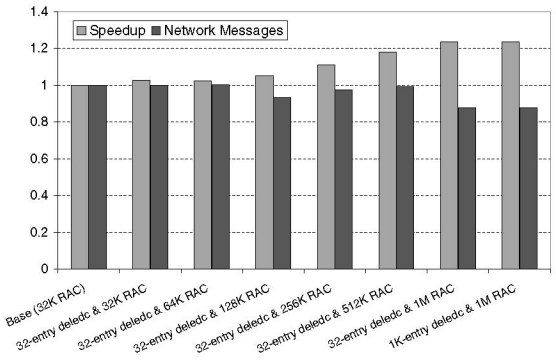

Figure 12. Sensitivity to RAC size (Appbt)

delegate cache size. MG has a substantial amount of producer-consumer data, so increasing the delegate cache size improves performance. For Appbt, the performance benefit of delegation/updates is limited by the size of the RAC; increasing the RAC size eliminates this bottleneck and improves performance.

4 Related Work

Adaptive coherence protocols have been proposed that optimize migratory [10, 32], pairwise [17], and wide sharing [18]. Prefetching can also be used to hide long miss latencies [6, 30], although overly aggressive prefetching can increase network traffic and pollute caches. Recent research has focused on developing weak/release-consistent producer-initiated mechanisms in which data are sent directly to consumer's caches [3, 19, 34, 5, 31, 27] .

A different approach is to support speculative coherence operations. Prediction in the context of shared memory was first proposed by Mukherjee and Hill, who show that it is possible to use 2-level address-based predictors at caches and directories to trace and predict coherence messages [29]. Lai and Falsafi improve upon these predictors by reducing history table sizes and showing how coalescing messages from different nodes can accelerate reads [20]. Alternatively, Kaxiras and Goodman propose instruction-based prediction for migratory sharing, wide sharing, and producer-consumer sharing [18].

Many research have proposed exploiting coherence prediction to convert 3-hop misses into 2-hop misses. Acacio *et al.* [4] and Martin *et al.* [24] propose mechanisms for predicting sharers. Lebeck and Wood propose dynamic self-invalidation [22], whereby processors speculatively flush blocks to reduce the latency of invalidations by subsequent writers. Lai and Falsafi propose a two-level adaptive predictor to more accurately predict when a cache line should be self-invalidated [21]. All of these techniques do a good job of predicting when a processor is done with a particular cache line, but all require changes to the processor die. Our simpler, and likely less accurate, predictor can be used in conjunction with unmodified commercial processors, since all changes are made at the external directory controller. As a result, our design can be adopted in near-future designs such as the next generation SGI Altix.

5 Conclusions and Future Work

Our work focuses on the design and analysis of mechanisms that improve shared memory performance by eliminating remote misses and coherence traffic. In this paper we propose two novel mechanisms, *directory delegation* and *speculative updates*, that can be used to improve the performance of applications that exhibit producer-consumer sharing. After detecting instances of producer-consumer sharing using a simple directory-based predictor, we delegate responsibility for the data's directory information from its home node to the current producer of the data, which can convert 3-hop coherence operations into 2-hop operations. We also present a speculative update mechanism wherein shortly after modifying a particular piece of data, producers speculatively forward updates to the nodes that most recently accessed it.

On a collection of seven benchmark programs, we demonstrate that speculative updates can significantly reduce the number of remote misses suffered and amount of network traffic generated. We consider two hardware implementations, one that requires very little hardware overhead (a 32-entry delegate cache and a 32KB RAC per node) and one that requires modest overhead (a 1K-entry delegate cache and a 1MB RAC per node). On the small configuration, delegation/updates reduce execution time by 13% by reducing the number of remote misses by 29% and network traffic by 17%. On the larger configuration, delegation/updates reduce program execution time by 21% by reducing the number of remote misses by 40% and network traffic by 15%. Finally, we show that the performance benefits derive

primarily from eliminating remote misses, and only secondarily from reducing network traffic.

There are many ways to extend and improve the work reported herein. To minimize the amount of hardware needed, thereby making it easier to adopt our ideas in near future products, we employ a very simplistic producer-consumer sharing pattern and "last write" predictor. Using a simple analytical model, not presented here for space reasons, we found that as network latency grows, the achievable speedup is limited to *1/(1-accuracy)*. Thus, we plan to investigate the value of more sophisticated predictors, e.g., one that can detect producer-consumer behavior in the face of false sharing and multiple writers. In addition, we plan to investigate the potential performance benefits of non-sequentially consistent versions of our mechanisms, e.g., ones that issued updates in place of invalidates rather than after invalidates, or that support multiple writers [7].

References

[1] http://www.top500.org/.

[2] http://phase.hpcc.jp/Omni/benchmarks/NPB/.

[3] H. Abdel-Shafi , J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fi ne-grain producer-initiated communication in cache-coherent multiprocessors. In *HPCA*, 1997.

[4] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *PACT*, 2002.

[5] C. Anderson and A. R. Karlin. Two adaptive hybrid cache coherency protocols. In *HPCA*, 1996.

[6] G. Byrd and M. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 1999.

[7] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[8] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing verifi cation complexity of a multicore coherence protocol using assume/guarantee. *FMCAD*, 2006.

[9] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *ISCA*, 2006.

[10] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[11] D. L. Dill. The Murphi verifi cation system. In *CAV*, 1996.

[12] D.H. Bailey, et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, Fall 1994.

[13] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. 2006.

[14] S. L. Graham, M. Snir, and C. A. Patterson. Getting up to speed: The future of supercomputing. In *Committee on the Future of Supercomputing, National Research Council*, 2004.

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

[16] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *ASPLOS*, 2004.

[17] Institute of Electrical and Electronics Engineers. *IEEE Standard for Scalable Coherent Interface: IEEE Std. 1596-1992*. 1993.

[18] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *HPCA*, 1999.

[19] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *ICS*, 1995.

[20] A.-C. Lai and B. Falsafi . Memory sharing predictor: The key to a speculative coherent DSM. In *ISCA*, 1999.

[21] A.-C. Lai and B. Falsafi . Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA*, 2000.

[22] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *ISCA*, 1995.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. 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.

[24] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared memory multiprocessors. In *ISCA*, 2003.

[25] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *ISCA*, 2003.

[26] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving multiple-cmp systems using token coherence. In *HPCA*, 2005.

[27] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. 2006.

[28] K. L. McMillan. Parameterized verifi cation of the FLASH cache coherence protocol by compositional model checking. In *Proceedings of the 11th Conference on Correct Hardware Design and Verifi cation Methods*, 2001.

[29] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *ISCA*, 1998.

[30] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[31] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi , and A. Moshovos. Spatial memory streaming. *ISCA*, 2006.

[32] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[33] K. Strauss, X. Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and fi ltering of snoops in embedded-ring multiprocessors. *ISCA*, 2006.

[34] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi . Temporal streaming of shared memory. In *ISCA*, 2005.

[35] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.