Appears in the *Proceedings of the 26th International Symposium on Computer Architecture (ISCA 26)*, 1999.

Memory Sharing Predictor: The Key to a Speculative Coherent DSM

An-Chow Lai and Babak Falsafi
School of Electrical & Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
impetus@ecn.purdue.edu, http://www.ece.purdue.edu/~impetus

Abstract

Recent research advocates using general message predictors to learn and predict the coherence activity in distributed shared memory (DSM). By accurately predicting a message and timely invoking the necessary coherence actions, a DSM can hide much of the remote access latency. This paper proposes the *Memory Sharing Predictors* (*MSPs*), pattern-based predictors that significantly improve prediction accuracy and implementation cost over general message predictors. An MSP is based on the key observation that to hide the remote access latency, a predictor must accurately predict only the remote memory accesses (i.e., request messages) and not the subsequent coherence messages invoked by an access. Simulation results indicate that MSPs improve prediction accuracy over general message predictors from 81% to 93% while requiring less storage overhead.

This paper also presents the first design and evaluation for a speculative coherent DSM using pattern-based predictors. We identify simple techniques and mechanisms to trigger prediction timely and perform speculation for remote read accesses. Our speculation hardware readily works with a conventional full-map write-invalidate coherence protocol without any modifications. Simulation results indicate that performing speculative read requests alone reduces execution times by 12% in our shared-memory applications.

1 Introduction

Distributed shared memory (DSM) is emerging as the architecture of choice for medium- to large-scale enterprise multiprocessor servers. DSMs offer programming compatibility with respect to the ubiquitous bus-based symmetric multiprocessors (SMPs) by providing a logical shared address space over physically distributed memory. DSMs also enhance scalability by removing the shared bus bottleneck in SMPs. Performance tuning applications on DSMs, however, can often be difficult due to the non-uniform nature of memory accesses. DSMs suffer from a lack of *performance transparency* with respect to SMPs because remote shared-memory accesses inherently take up to ten to a hundred times longer than local memory accesses.

To address this issue, aggressive DSM implementations directly target reducing the remote access latency [14]. These designs repackage processors into custom motherboards with fully integrated DSM memory controllers and custom interconnects. Requiring custom motherboards, however, prevents these DSMs from exploiting the excellent cost-performance of off-the-shelf desktops and server motherboards. Moreover, current aggressive DSMs at best reduce the remote access latency to two or three times local access latency, leaving a large remote-to-local access performance gap.

Other proposals for improving DSM performance include techniques to reduce remote access frequency [8,10], hide or tolerate remote access latency [1,2], or reduce the coherence protocol overhead [11,15,14,7,13]. Many such techniques are non-transparent and require either careful annotation by the application programmer or complex compiler analysis. Transparent techniques often have limited applicability and only work well for regular memory access patterns or target specific sharing patterns known a priori. Techniques to reduce coherence overhead also typically rely on complex adaptive coherence protocols which directly capture the sharing patterns in the protocol states. Such protocols use complex finite-state-machines which are difficult to design and require large amounts of computational resources to verify [6]. Moreover, capturing sharing patterns in protocol states often limits the protocol to learning one sharing pattern per memory block at a time.

In a recent paper [17], Mukherjee and Hill advocate using a general pattern-based predictor—derived from Yeh and Patt's two-level adaptive *Pap* branch predictor [23]—to learn and predict the coherence activity for a memory block in a DSM. By accurately predicting and performing the necessary coherence operations speculatively in advance, a predictor-based DSM can potentially eliminate all of the coherence overhead, resulting in remote accesses that are as fast as a local access. Such a predictor is based on the key observation that much as branches tend to have a repetitive nature (e.g., backwards branches are often taken because loops iterate) leading to accurate predictability, memory blocks often have a small number of stable, repetitive, and predictable sharing patterns [9].

A general pattern-based predictor is in many respects superior to an adaptive coherence protocol. A pattern-based predictor can dynamically learn and adapt to an applications's sharing patterns at runtime. Moreover, a predictor is capable of simultaneously capturing multiple dis-

tinct sharing patterns for a given memory block. Finally, a predictor merely provides hints to perform coherence operations early, obviating the need to modify the base coherence protocol.

This paper proposes novel pattern-based predictors, *Memory Sharing Predictors* (*MSPs*), that dramatically improve prediction accuracy and implementation cost over previous proposals. Unlike a general message predictor [17], an MSP only predicts memory request messages—i.e., the primary messages that invoke a sequence of protocol actions. In common DSM sharing patterns, multiple coherence messages in a read-sharing phase often arrive in an arbitrary order due to system contention or load imbalance in the application. By eliminating the acknowledgment messages from the pattern tables, MSPs substantially reduce perturbation in the tables due to message re-ordering, reduce the predictor's memory overhead, and significantly increase prediction accuracy.

We present simulation results running shared-memory applications to indicate that:

- *MSP*, our base predictor, improves prediction accuracy in a general message predictor from 81% to 86% by eliminating the acknowledgment messages from the pattern tables,
- *VMSP*, our optimized predictor, additionally improves prediction accuracy to 93% by using a compact vector representation of read sequences thereby eliminating read request re-ordering,
- VMSP not only offers the best prediction accuracy but also reduces implementation cost in terms of storage overhead over a general message predictor,

In this paper, we present the first design for a speculative coherent DSM using pattern-based predictors. To successfully hide the remote access latency, a speculative coherent DSM must accurately predict both "what" memory requests subsequently arrive and "when" they arrive. We primarily focus on executing coherence operations speculatively to hide the remote read latency. Our MSPs use two techniques to trigger speculation for read requests timely. We use a simple *Speculative Write Invalidation* (*SWI*) heuristic which predicts when a producer is done writing to a memory block, invalidates the writable copy speculatively, and forwards the block to the consumers. When SWI fails to invalidate writable blocks early, we use the read request from the first consumer to trigger speculation and forward the block to the rest of the consumers.

Results from a simple analytic model and simulation of a speculative coherent DSM indicate that:

- high-accuracy predictors are the key to high performance in a speculative coherent DSM,
- triggering read request speculation for a read sequence based on the arrival of the first read reduces execution time in all applications on average by 8% and at best by 17%,
- triggering speculation using SWI reduces execution time on average by 12% and at best by 24%.

In the following section, we describe the anatomy of DSM coherence protocols and general message predictors. In Section 3, we introduce our memory sharing predictors. Section 4 describes our design for a speculative coherent DSM. Section 5 characterizes the key factors impacting performance and presents a qualitative performance analysis using a simple analytic model. Section 6 and Section 7 present the simulation methodology and results. Finally, Section 8 presents a summary and concludes the paper.

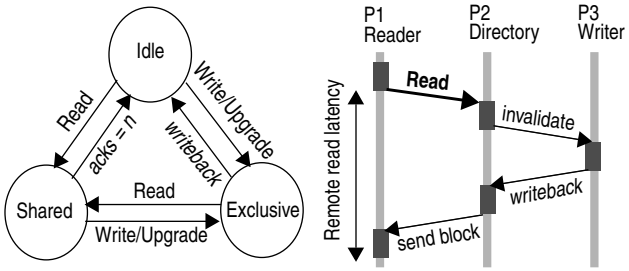

FIGURE 1: Directory protocol transitions (left) and example sequence of protocol operations on a remote read request (right).

2 Background

DSM allocates and distributes memory pages across the machine nodes. On every node, a directory maintains sharing information for the memory pages (also referred to as home pages) designated to that node. For every fine-grain (e.g., 32-128 byte) memory block on a home page, the directory maintains a block sharing state and a list of processor ids sharing the block. A coherence protocol coordinates sharing of memory blocks among the processors.

For every memory block, the protocol implements a finite-state-machine in which a state corresponds to the directory state for the block and actions are messages sent over the network to coordinate sharing. In this paper, we study simple full-map write-invalidate coherence protocols implemented in hardware such as those in SGI Origin 2000 [14], Sequent NUMA-Q [16], and Sun WildFire [10]. The ideas we present, however, are also applicable to other implementations such as fine-grain software [21] and firmware [19], as well as page-based DSM [3].

Figure 1 (left) illustrates the state machine for a simple invalidation-based coherence protocol. A memory block is either in the *Idle* state indicating that there are no (remote) processors with valid copies of the block, in a (read-) *Shared* state indicating that one or more processors have a read-only copy of the block, or in the *Exclusive* state indicating that a single processor owns a writable copy of the block. There are three types of memory access requests. A *read* is a request to fetch a read-only copy of a block. A *write* is a request to obtain a writable copy of a block. An *upgrade* is a request to write to an already cached read-only copy of the block.

Figure 1 (right) illustrates an example sequence of coherence actions when a processor requests a read-only copy of a block. The directory first invalidates and requests a writeback for the current writable copy of the block, and subsequently sends a read-only copy to the requesting processor. The entire read transaction includes four network messages and up to four local memory accesses making remote access latencies much higher than local memory latencies.

To transparently reduce the remote memory access latency, a speculative coherent DSM must *accurately* pre-

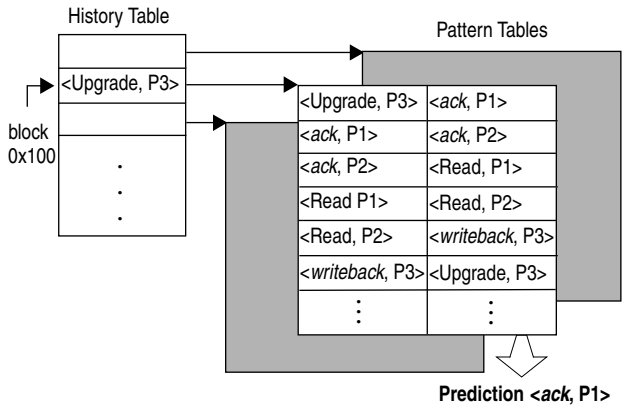

FIGURE 2: A two-level message predictor.

dict the remote access and *timely* perform the necessary coherence actions in advance. For instance, in Figure 1 (right) if the DSM hardware at P2 accurately and timely predicts the read access by P1, it can invalidate and forward the block to P1 well in advance to hide the entire latency of the remote read.

2.1 Pattern-Based Message Predictors

A pattern-based coherence message predictor is derived from the widely-used two-level adaptive *PAp* branch predictor [23]. Figure 2 depicts the anatomy of a two-level message predictor capturing message sequences for memory blocks at the directory. A history table records the most recent sequence of incoming coherence messages for every memory block. A pattern table records all observed sequences of coherence messages for every memory block. An entry in the pattern table consists of a message sequence and a *prediction* for the subsequent message given the sequence. The prediction is the observed immediate successor of the message sequence when the sequence last occurred.

The predictor in the figure has a history depth of one—i.e., the predictor maintains a history of the most recent coherence message for every block. The figure illustrates an example of possible message sequences for a simple producer/consumer sharing among three processors. Request messages appear capitalized and protocol acknowledgement messages appear in italics. Processor 3 (P3) writes to the memory block at address 0x100 and processors 1 (P1) and 2 (P2) subsequently read the block. The protocol receives an upgrade request (recorded in the history table) by P3 and is in the process of invalidating the read-only copies of P1 and P2. The pattern table predicts the next incoming message given the specific sequence to be an acknowledgment by P1. The acknowledgment is in response to an invalidation sent by the directory to P1.

The predictor's performance and cost are both highly sensitive to the history depth. Much as in branch predictors, a deeper history enables the predictor to be more selective by distinguishing among message sequences with common patterns. Such sequences may result from true application sharing patterns. For instance, assume in our example that P3 and P2 alternate upgrading the block. As before, an upgrade from P3 would be followed by reads from P1 and P2. Similarly, an upgrade from P2 would be followed by reads from P1 and P3. For such a sharing pattern, the predictor in the figure would always mispredict the writer because a history depth of one prevents the predictor from distinguishing between the write-backs from P3 and P2. A history depth of two would include both readers and allow the predictor to distinguish between the writers.

Race conditions in message arrivals also result in different message sequences for the same sharing pattern. Message re-ordering in the network or queueing delays at the directory or caches may result in race conditions among the messages. For instance, in our example, P2 and P3 may simultaneously request the memory block but the messages may arrive in an arbitrary order at the directory. A predictor with a history depth of one would fail to predict accurately either the read or the upgrade requests if read request messages were frequently re-ordered. In contrast, a predictor with a history depth of two would learn both possible re-orderings of the reads and predict both the reads and the upgrade accurately.

Although a larger history improves the prediction accuracy, it may prohibitively increase the predictor's cost [17]. In the limit, the number of pattern table entries is directly proportional to the history depth. In practice, memory blocks exhibit a small number of stable and distinct sharing patterns [9]. Consequently, in the absence of message re-ordering, a memory block would require a small number of pattern table entries independent of the history depth. In the worst case, however, message re-ordering increases the required number of pattern table entries by the permutation of all possible re-orderings.

A DSM may directly implement the history table within the directory because of the fixed amount of required storage for a history entry. The required size of the pattern tables directly depends on a memory block's sharing activity which may largely vary among blocks. In this paper, we assume the same table allocation and implementation strategies as discussed in [17].

3 Memory Sharing Predictors (MSPs)

This paper proposes a new class of pattern-based predictors called the *Memory Sharing Predictor*s (MSPs). An MSP is based on the key observation that to eliminate the coherence overhead on a remote access latency it is only necessary to predict the memory request messages (i.e., a read, write, or an upgrade). A general message predictor unnecessarily predicts the coherence acknowledgement messages (i.e., an invalidation response or a writeback) as well, even though these messages are in direct response to a coherence action and are always expected to arrive. In Figure 1 (right), the writeback message by P3 is in direct response to the invalidation message by P2. The writeback is only a response to the coherence activity invoked by the read request and is itself part of the coherence overhead.

Because it predicts all coherence messages, a general message predictor has several key shortcomings. First, since the protocol overlaps the invalidation messages for a block, the acknowledgments may arrive in any arbitrary order. Predicting acknowledgments may unnecessarily and severely perturb prediction of the (more fundamental) request messages if acknowledgments often arrive out of order. Second, predicting the acknowledgments unneces-

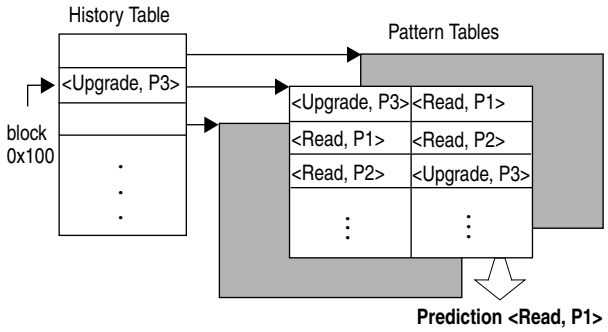

FIGURE 3: A Memory Sharing Predictor (MSP).

sarily increases the number of pattern table entries. Third, predicting the acknowledgments increases the required number of bits to encode message types in both the history and pattern tables.

MSP addresses the above shortcoming in a general message predictor by only predicting the request messages. Figure 3 illustrates the anatomy of an MSP. As compared to the message predictor in Figure 2, the MSP eliminates half of the pattern table entries in our example of producer/consumer sharing. The MSP would also eliminate all the sequences resulting from the potential reorderings of the acknowledgments (not shown in the figure). The MSP requires two bits to encode three request message types (i.e., read, write, and upgrade) as compared to a message predictor requiring three bits to encode three request types and two acknowledgement types (i.e., responses to read-only invalidations and writebacks).

3.1 VMSP: Using Vectors to Encode Reads

We further refine the MSP design and propose the *Vector MSP* (VMSP). VMSP is based on the primary observation that because a full-map protocol allows multiple processors to simultaneously cache a read-only copy of a memory block, a predictor must simply identify the readers and need not maintain the order in which they read. Rather than record and predict the read requests as individual pattern table entries as in MSP, VMSP encodes a sequence of read requests in a bit-vector much as a full-map directory maintains the identity of multiple readers of a block. Figure 4 illustrates the anatomy of a two-level VMSP. Compared to MSP, VMSP reduces the number of pattern table entries required to capture our example sharing pattern from three to two.

VMSP's key advantage over MSP is that by not maintaining the order among the reads, VMSP eliminates the negative impact of read request re-ordering. For example in our MSP from Figure 3, a re-ordering of read requests from P2 and P3 would result in a misprediction in all the pattern table entries. MSP requires a history depth of at least two to simultaneously learn and capture both possible re-orderings of the reads. In general, for n readers, MSP requires a history depth of n to capture all possible reorderings of the read requests. As such, the number of required pattern table entries can quickly grow with the number of readers. VMSP folds all the readers into a single vector thereby substantially reducing the number of pattern table entries.

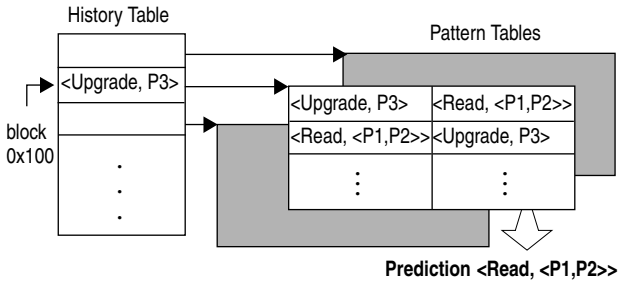

FIGURE 4: A Vector MSP (VMSP).

VMSP, however, increases the minimum required number of bits to encode a read sequence as compared to MSP. VMSP uses two bits to encode the read request type and n bits to encode the list of readers for a machine with n processors. In contrast, MSP only requires two bits for the type and log(n) bits to encode a processor id for every read request. Therefore, VMSP only offers a more compact encoding if the actual number of readers is greater than (2+n)/(2+log(n)). To break even with MSP in the encoding size, VMSP requires at least two readers per block for a machine with eight processors and at least three readers per block for a machine with sixteen processors respectively.

4 Mechanisms for a Speculative DSM

A speculative coherent DSM requires three primary mechanisms to hide the remote access latency successfully: (1) a mechanism to predict "what" memory requests subsequently arrive, (2) a mechanism to predict "when" subsequent remote accesses arrive, and (3) a mechanism to execute the necessary coherence operations for a predicted remote access speculatively. While our pattern-based predictors only predict what subsequent remote accesses arrive, they fail to predict when they arrive. In this section, we identify and propose simple techniques to predict requests timely, and describe mechanisms to execute the necessary coherence actions speculatively. In the next section, we present a qualitative analysis for the performance of a speculative coherent DSM using a simple analytic model. In Section 7, we evaluate performance using empirical results from detailed simulations.

Figure 5 illustrates the anatomy of a speculative coherent DSM node. The node consists of one or more processors with their cache hierarchy. The processors are interconnected either via a snoopy bus to memory and a DSM board [16,10], or through a switch tightly integrating the DSM hardware with the memory controller [14]. The DSM hardware implements the coherence protocol, and includes a remote cache (i.e., as a large repository for remote data) and a directory to maintain sharing information for the node's home pages. The hardware also includes an MSP to predict and execute coherence operations speculatively.

4.1 Triggering Request Speculation

The success of a speculative coherent DSM relies on the accuracy of mechanisms to execute the necessary coherence actions for a remote access timely. Much like mispredicting a remote request, premature coherence

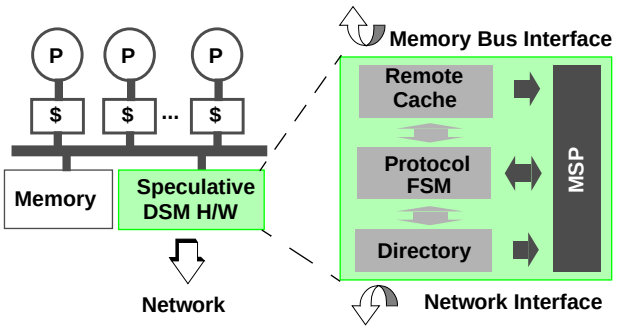

FIGURE 5: A speculative coherent DSM node (left) and the coherence hardware (right).

speculation can result in a significant increase in remote access latencies. For instance, early speculation on a write may prematurely take a block away from its readers. Similarly, late coherence speculation may limit a DSM's ability to hide much of the coherence overhead and offset the gains from speculation.

A request arrival time, to the first order, is a function of an application's memory access patterns. While a block's sharing patterns at the directory can be captured using an MSP, sharing does not provide information about *when* a processor accesses a specific block. Proposals for hardware prefetching have extensively focused on learning and predicting an application's memory access patterns. These techniques, however, are either only effective for regular access strides [4], target irregular accesses for specific data structures and have limited applicability [20], and are not as effective for arbitrary access patterns [12]. Rather than learn and predict access patterns, a DSM can rely on the observed coherence activity and message traffic to estimate a request's arrival time. Message traffic, however, is highly dependent on the amount of system contention and may lead to high inaccuracies in estimates.

Fortunately, there are common DSM sharing patterns that give rise to trigger-ready speculation—i.e., a speculation that may readily invoke the protocol. Read-sharing by more than one processor results in a trigger-ready speculation. In a predicted sequence of reads, the arrival of the first read may readily trigger speculation for the rest of the sequence. Similarly, migratory sharing results in a trigger-ready speculation. Migratory sharing is characterized by read and upgrade request pairs to a block by a given processor. When predicting migratory sharing, the arrival of the read by the processor may readily trigger speculation for the upgrade.

Moreover, there are common memory access patterns that may be predicted using simple heuristics. For instance, in many producer/consumer sharing scenarios, a producer often writes to a memory block only once and no longer accesses the block until the consumers read the new data. Such a sharing pattern is common in parallel commercial database servers which use message buffers to communicate information among processes. Rather than predict when the read requests from the consumers arrive, a DSM can predict when the producer has completed writing to the memory block.

In this paper, we propose a simple heuristic, called *Speculative Write-Invalidation* (SWI), in which an MSP predicts that a processor is done writing to a memory block upon a subsequent write (or upgrade) request to another block by the same processor. The MSP maintains an early-write-invalidate table recording the block address of the last write (or upgrade) request per processor. SWI not only hides the write invalidation latency, but also enables triggering speculation for the consumers' read requests. In the best case, both the write invalidation and the read request latency for all the consumers are eliminated.

While SWI is an excellent simple mechanism to trigger speculative read requests, it relies on incoming write requests and the subsequent invalidation to predict when to trigger the reads. As such, SWI precludes speculatively executing write (or upgrade) requests and a more general mechanism is required to trigger timely both read and write speculation. This paper is a first step towards implementing a speculative coherent DSM. As such, we primarily focus on executing reads speculatively. Our DSM triggers a sequence of reads upon a successful write invalidation using SWI or upon receiving the first read request in a sequence of reads when SWI fails.

4.2 Speculative Coherence Operations

The final enabling technology for a speculative coherent DSM are mechanisms to execute a coherence action speculatively and update the predictor's accuracy by verifying the speculation (i.e., verifying that the predicted access occurs). The key requirement for these mechanisms is that they co-exist with the base coherence protocol without any needed protocol modifications. Rather than require extra functionality in the protocol, the MSP simply advises the protocol to execute (existing) coherence operations early. A misspeculation results in additional (base) protocol transitions but does not interfere with the protocol's functionality. For instance, a premature write invalidation simply results in an extra subsequent read or write request by the producer processor.

To execute read requests speculatively, an MSP simply advises the protocol to send read-only block copies to the predicted requesters. To verify the speculation accuracy, the DSM uses a reference bit in the remote cache for every block that is placed speculatively. Upon a reference from a processor, the remote cache clears the bit verifying that the speculated access occurs. When blocks are invalidated from the remote cache, the speculative bit is piggy-backed on the invalidation message sent to the home node. The MSP (at the home node) determines the speculation accuracy using the piggy-back information, and removes mispredicted request sequences from the pattern tables. To obviate the need for protocol modification, upon a race between a speculatively-sent block and an in-flight read request for the block, the DSM node receiving the block drops the speculated message and awaits a response to the read request message from the protocol.

When MSP predicts a sequence of reads upon receiving a write (or upgrade), it uses SWI to simply advise the protocol to send a write invalidation. A successful invalidation triggers speculation for the read sequence. To prevent frequent premature invalidations, SWI uses a bit per write (or upgrade) in the corresponding pattern table entry indicating a previous premature invalidation for the write. For read sequences that follow a write (or upgrade) which SWI

no longer invalidates, the read speculation can only proceed upon receiving the first read request.

5 A Qualitative Performance Analysis

Much like speculative instruction execution using branch prediction, the performance of a speculative coherent DSM depends on the speculation (or prediction) accuracy, reduction in latencies upon a successful speculation, and the misspeculation penalty. Unlike speculative instruction execution, the performance of a speculative coherence protocol depends on the opportunity for speculation. A computation-intensive application, for instance, is unlikely to benefit from hiding remote access latencies. In this section, we present a simple intuitive model to analyze the performance of a speculative coherent DSM.

Our analytic model captures the key factors affecting performance in a speculative coherent DSM in a small number of parameters. The model estimates performance improvement by accounting for the reduction in communication time on the execution's critical path. Our model makes several simplifying assumptions. We assume that when the DSM successfully executes a speculative memory request, the entire remote latency is hidden. We also assume a misspeculation only slows down a remote access and does not increase the request frequency. In general, however, a speculative coherent DSM can incorrectly take a block away from a current user, thereby turning a potential processor cache hit into a much slower remote access latency. The model, however, can approximate such an increase in the request frequency as a higher overall misspeculation penalty.

Our performance model includes the following parameters: c is the application's communication ratio on the critical path, f is the fraction of speculatively-executed memory requests over all the received requests, p is the request prediction accuracy, l_{access} and r_{access} represent the local and remote memory latencies respectively, rtl is the ratio of remote to local access latencies, n is the misspeculation penalty factor, and N is the number of remote requests on the critical path.

$$\text{comm-speedup} = \frac{\text{communication time without speculation}}{\text{communication time with speculation}}$$

$$= \frac{N\, r_{access}}{(1-f)N\, r_{access} + f\, N\, (p l_{access} + (1-p)\, n\, r_{access})}$$

$$= \frac{1}{(1-f) + f\, \left(\frac{p}{rtl} + n\,(1-p)\right)} \qquad \text{(EQ 1)}$$

$$\textbf{speedup} = \frac{\text{total execution time without speculation}}{\text{total execution time with speculation}}$$

$$= \frac{1}{(1-c) + \frac{c}{\text{com-speedup}}} \qquad \text{(EQ 2)}$$

$$= \frac{1}{(1-c) + c\left((1-f) + f\left(\frac{p}{rtl} + n\,(1-p)\right)\right)}$$

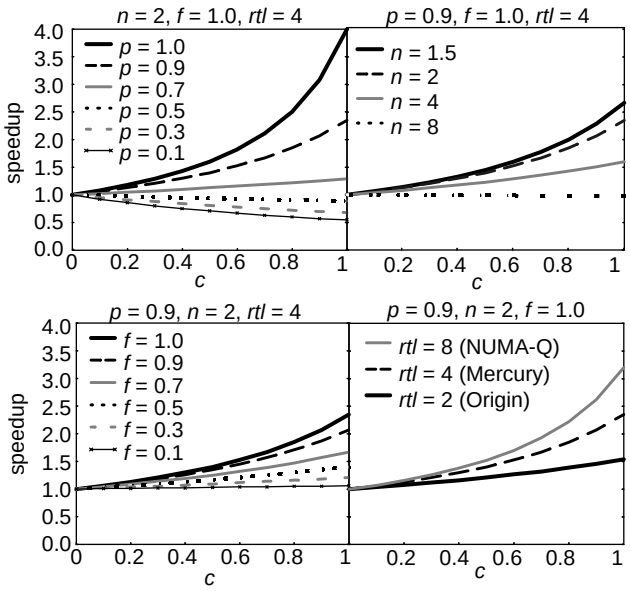

FIGURE 6: Potential speedup in a speculative coherent DSM.

The model approximates the communication time in a conventional DSM by $N\, r_{access}$. In a speculative coherent DSM, a fraction f of memory requests execute speculatively, out of which p succeed and convert the remote access into a local one incurring a latency of l_{access} instead. (1-p) of the speculative accesses fail and result in a misspeculation penalty of $n\, r_{access}$. Equation 1 depicts the resulting speedup in communication time. Equation 2 estimates the overall application speedup by reducing the communication time by the speedup factor from Equation 1.

Figure 6 examines the potential for speedup using a speculative coherent DSM. The graphs plot speedup of a speculative coherent DSM from Equation 2 against application communication ratio, c. The graphs at the top-left examine the impact of prediction accuracy on speedup for a DSM with a moderate remote-to-local latency ratio of 4 (characteristic of today's aggressive DSM clusters [22]) and a misspeculation penalty factor of 2.

The graphs corroborate the intuition that prediction accuracy plays a primary role in performance. A low prediction accuracy of 10%-50% consistently results in a slowdown due to a high misspeculation overhead. A prediction accuracy of 70% at best speeds up the execution by 25% for a fully communication-bound application, while a higher prediction accuracy of 90% improves performance even for applications with moderate communication ratios. In the limit, when all speculations succeed (p=1.0), all remote accesses in the speculative coherent DSM turn into local accesses and the DSM behaves like an SMP—i.e., a uniform memory architecture. These results indicate that designing accurate predictors is a key first step in building speculative coherent DSMs. We present empirical results in Section 7 that indicate that our proposed MSPs significantly improve prediction accuracy over current predictors.

A misspeculation can vary from merely sending a read-only copy of a block to a non-reader during the read phase

Number of nodes	16
Processor speed	600 MHz
Processor cache	1 Mbyte
Memory bus	100 MHz
Local memory/ Remote Cache access time	104 cycles
Network latency	80 cycles
Round-trip miss latency	418 cycles
Remote-to-local access ratio (rtl)	~4

Table 1: System configuration parameters.

Application	Input Data Sets	Iterations
appbt	12x12x12 cubes	40
barnes	4K particles	21
em3d	76800 nodes, 15% remote	50
moldyn	2048 particles	60
ocean	130x130 array	12
tomcatv	128x128 array	50
unstructured	mesh.2K	50

Table 2: Applications and input data sets.

requiring an extra invalidation, to taking a block incorrectly away from a processor actively accessing the block, converting processor cache hits to remote accesses. If misspeculations are infrequent, however, the penalty does not have a large impact on performance. Figure 6 (top-right) examines the impact of misspeculation penalty on performance. The graphs corroborate this intuition and indicate that performance is not as sensitive to misspeculation penalty at a high perdition accuracy. Speedups improve with increasing communication ratio even with a misspeculation penalty factor of 4 times a remote access latency.

There are several factors affecting the fraction of speculatively-executed requests, *f*. The type of requests executed speculatively has a primary impact on *f*. A predictor's learning speed—i.e., the number of request message it takes to learn and predict a message sequence—is also a key factor affecting *f*. The higher the history depth, the longer it takes the predictor to learn a new message sequence. In other words, *f* is a measure of the reuse frequency for pattern table entries. Applications with rapidly-changing sharing patterns may frequently introduce new pattern table entries without reusing them. Figure 6 (bottom-left) plots speedup with varying values for *f*. The graphs indicate that a low fraction of speculatively-executed requests—e.g., as a result of rapidly-changing sharing patterns—will fundamentally limit performance even with high prediction accuracies.

Finally, Figure 6 (bottom-right) examines the impact of remote-to-local latency ratio (*rtl*) on speedups. The graphs plot speedups for minimum *rtl* values found in recent designs such as the tightly-coupled high-end SGI Origin 2000 [14] and two more cost-effective cluster-based DSMs, the HAL Mercury [22] and the Sequent NUMA-Q [16]. The graphs indicate that while a speculative coherence protocol benefits Origin, it benefits the clusters most due to a much higher remote-to-local access ratios. This result also indicates that a speculative coherent DSM architecture may help eliminate the performance gap between the clusters and the high-end system, enabling the clusters to offer equal performance at a much lower cost.

6 Methodology

To evaluate practical implementations of a speculative coherent DSM, we use the Wisconsin Wind Tunnel II [18] to simulate a CCNUMA with sixteen nodes interconnected through hardware DSM boards to a low-latency switch-based network (Figure 5). Table 1 depicts the system configuration parameters for the simulated DSM. Each node contains a 600-MHz dual-issue processor with 1-Mbyte caches interconnected by a 100 MHz split-transaction bus to memory and the DSM board. We assume perfect instruction caches but model data caches and their contention at the memory bus accurately. We further assume a point-to-point network with a constant latency of 80 cycles but model contention at the network interfaces.

Recent aggressive caching techniques have proven to virtually eliminate all of capacity and conflict request traffic resulting from a node's inability to simultaneously hold all the necessary remote data [8,10]. Rather than inflate results with unnecessary communication, we gauge a speculative coherent DSM's ability to hide true communication latency and assume a remote cache large enough to hold the remote data. We model a full-map write-invalidate protocol using 32-byte coherence blocks.

Table 2 presents the applications we use in this study and the corresponding input parameters. *Appbt* is a shared-memory implementation of the NAS benchmark. *Barnes* and *ocean* are from the SPLASH-2 benchmark suite. *Em3d* is a shared-memory implementation of the Split-C benchmark. *Moldyn* is a shared-memory implementation of a CHARMM-like molecular dynamics application (similar to the one used in [17]). *Tomcatv* is a shared-memory implementation of the SPEC benchmark. *Unstructured* is a computational fluid dynamics application that uses an unstructured mesh. Our shared-memory implementation of *unstructured* uses a cyclic partitioning algorithm for the mesh and is therefore more communication-intensive than optimized implementations using the recursive coordinate bisection partitioning algorithm [17].

7 Results

The results from our simple analytic model clearly indicate that high prediction accuracy is fundamental to successfully performing coherence speculation. In this section, we first compare our proposed memory sharing predictors (MSP and VMSP) to a previously proposed coherence message predictor called Cosmos [17]. We present simulation results indicating that MSP and VMSP significantly improve prediction accuracy over Cosmos. Next, we show that our predictors also reduce implementation cost as compared to Cosmos. We also show that despite the higher prediction accuracy, MSP and VMSP also offer a competitive learning speed as compared to

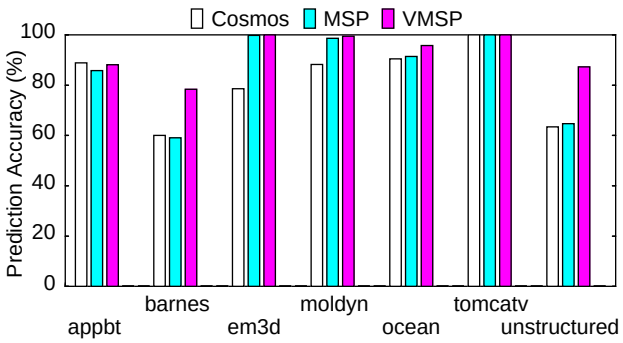

FIGURE 7: Base predictor accuracy comparison.

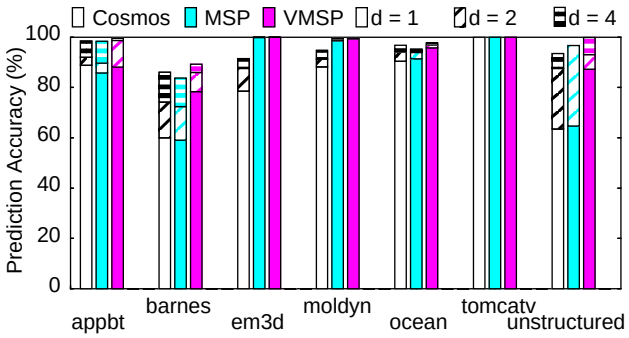

FIGURE 8: Predictor accuracy with varying history depth (d).

Cosmos. Finally, we present numbers evaluating the performance of the first proposal for a speculative coherent DSM.

7.1 Predictor Accuracy

Figure 7 compares the prediction accuracy in Cosmos, MSP, and VMSP, for a history depth of one. The figure plots the number of correctly predicted messages over the total number of predicted messages. The figure indicates that prediction accuracy in Cosmos is higher than 90% in only two out of the seven applications. Furthermore, in another three applications the accuracy is lower than 80%, and in the worst case the accuracy is as low as 60%. MSP and VMSP increase prediction accuracy to significantly higher levels as compared to Cosmos by eliminating the perturbation in the pattern tables due to the protocol acknowledgements. MSP's accuracies are either comparable to or much higher (by an additional 15%-20%) than Cosmos. VMSP also eliminates the read request re-ordering, performs best, and increases accuracy to over 87% in all but one application, and over 79% in all applications.

Em3d exhibits producer/consumer sharing with a small read-sharing degree, and reaches a 99% accuracy with MSP alone as compared to Cosmos.

Moldyn and *unstructured* exhibit both producer/consumer and migratory sharing. In *moldyn*, the producer/consumer phase exhibits a small read-sharing degree and is highly predictable even with MSP. The migratory sharing patterns remain static throughout the application and are also highly predictable. As a result, both MSP and VMSP significantly improve prediction accuracy over Cosmos and reach an accuracy of 98%-99%.

Unstructured exhibits wide read-sharing in a producer/consumer phase. In this phase, all the processors read a block resulting in high read request re-ordering in MSP and an accuracy of under 65%. VMSP removes the read re-ordering and substantially improves the prediction accuracy in the producer/consumer phase. The migratory sharing in *unstructured* occurs in a sum reduction phase. To optimize communication, processors refrain from reading/writing memory blocks if their contribution to the sum is a zero. Some processors compute a zero in every other visit to the reduction, and therefore alternate participating in the migratory sharing. As such, with a history depth of one, the predictors both mispredict the processors in the migratory sharing and the subsequent consumers (in the producer/consumer phase) upon leaving the reduction. The resulting mispredictions limit VMSP's accuracy to 87%.

Ocean and *tomcatv* are both stencil computations in which processors only communicate with their immediate neighbors and there is only a single consumer per block. All three predictors reach a 100% accuracy for *tomcatv*. *Ocean*, however, uses a lock to implement a reduction and sum a value over all processors at the end of every iteration. The order in which processors enter the lock changes every iteration reducing VMSP's prediction accuracy to slightly below 100%.

Appbt implements a gaussian elimination over a cube in which processors are allocated subcubes and share boundary values on the subcube surfaces. Because the gaussian elimination proceeds in all three cube dimensions in subsequent steps, the memory blocks located at a subcube edge are consumed by two different processors along two different dimensions [5]. With a history depth of one, all predictors fail to distinguish sharing along the different cube dimensions for the blocks at the subcube edges resulting in a prediction accuracy of at best 90%. Cosmos slightly improves accuracy over MSP because acknowledgement messages in *appbt* actually help distinguish between read sharing along the different dimensions.

Barnes simulates the forces among the bodies in a galaxy. In each iteration, processors traverse an octree representing the galaxy to calculate forces between the bodies. Most of the time a processor only partially traverses the octree to compute the forces for a given body. In every iteration, the tree is rebuilt to reflect the movement of bodies in the galaxy and this results in rapid changes in read-sharing patterns. While there is read-sharing by more than one processor on the octree, it does not result in a re-ordering of acknowledgments because the read-sharing is asynchronous, and there is minimal queueing in the system. As such, the acknowledgments arrive in the same order every time, and MSP does not improve accuracy over Cosmos. The readers, however, do not arrive in the same order in every iteration because a processor's workload during the tree traversal changes with a change in the octree structure. VMSP eliminates the re-ordering of reads in the pattern tables and increases accuracy to slightly less than 80%.

7.2 History Depth

A key advantage of pattern-based predictors over adaptive protocols is the predictors' ability to capture simultaneously multiple sharing patterns for a given memory

Application	Cosmos			MSP			VMSP		
	d=1	d=4	d=1	d=1	d=4	d=1	d=1	d=4	d=1
	pte	pte	ovh	pte	pte	ovh	pte	pte	ovh
appbt	5	8	10	3	5	6	2	3	9
barnes	11	42	21	7	25	11	5	12	18
em3d	5	21	10	4	4	6	2	2	8
moldyn	4	14	8	2	3	4	2	2	7
ocean	1	2	3	< 1	< 1	2	< 1	< 1	4
tomcatv	3	3	7	2	2	4	2	2	7
unstructured	9	168	17	5	8	8	4	4	14

Table 4: Predictor storage overhead.

block. Moreover, unlike adaptive protocols, the extent to which a predictor can capture multiple sharing patterns depends on the history depth and not the protocol complexity. A higher history depth enables the predictor to distinguish among distinct sharing patterns with common message sequences and increases the prediction accuracy. A higher history depth also changes the balance among the three predictors by allowing a predictor to capture multiple re-orderings of messages simultaneously thereby reducing the negative impact of re-ordering.

Figure 8 compares the prediction accuracy in Cosmos, MSP, and VMSP for history depths of one, two, and four. The graphs indicate that VMSP achieves higher accuracies for the applications which exhibit multiple sharing patterns with common message sequences, such as *appbt* and *unstructured*. A history depth of two enables the predictor to capture simultaneously the alternating sharing patterns in *appbt* for blocks at subcube edges, improving prediction accuracy to 100%. Similarly, with a larger history depth, the predictors can distinguish between the migratory sharing patterns in alternate reduction phases in *unstructured*, improving accuracy to up to 99%.

Because the structure of the octree in *barnes* rapidly changes, many blocks have message sequences with little or no reuse frequency. With a lower history depth, rapidly-changing sharing patterns will result in frequent mispredictions and a lower prediction accuracy. A higher history depth allows a larger number of sharing patterns to coexist, increases the learning time for the rapidly-changing sharing patterns, and reduces the overall prediction frequency. As such, only frequently occurring and more stable sharing patterns result in actual predictions, thereby increasing the prediction accuracy.

A key performance metric for a predictor, besides accuracy, is the speed at which it can learn and predict message sequences. For a given history depth, VMSP by nature is slower than Cosmos and MSP because the read sequence encoded in a single vector in VMSP may correspond to many pattern table entries in Cosmos and MSP. On the other hand, because VMSP significantly improves prediction accuracy over Cosmos and MSP, it may result in an overall larger number of correctly predicted messages.

Table 3 compares the predictors' learning speed. The table depicts the fraction of messages predicted by each predictor for a history depth of one. The table depicts the fraction of the messages correctly predicted as a product of prediction accuracy and the fraction of messages predicted. Besides *barnes* and *ocean*, the rest of the applications exhibit a high prediction frequency due to the highly iterative nature of the computation resulting in frequent reuse of the pattern table entries in the predictors. Moreover, MSP predicts the same number of messages as Cosmos whereas VMSP requires a slightly longer learning time. Nevertheless, VMSP's slower speed is offset by its much higher prediction accuracy resulting in an overall much larger number of correctly predicted messages.

Application	Cosmos (%)	MSP (%)	VMSP (%)
appbt	97 (87)	97 (83)	96 (85)
barnes	88 (53)	87 (52)	81 (63)
em3d	98 (77)	97 (97)	96 (96)
moldyn	97 (86)	97 (96)	97 (97)
ocean	89 (80)	86 (79)	83 (80)
tomcatv	97 (97)	97 (97)	95 (95)
unstructured	99 (63)	99 (65)	99 (87)

Table 3: Messages predicted (and correctly predicted) for a history depth of one.

7.3 Predictor Cost

A predictor's implementation cost is a direct function of the number of learned message sequences and the overhead of storing a message sequence in the pattern table. MSP and VMSP reduce the required number of pattern table entries by eliminating the acknowledgments from the pattern tables. VMSP further reduces overhead by eliminating the multiple re-orderings of a read sequence. On a per-entry basis VMSP, however, requires a higher storage overhead as compared to Cosmos and MSP because it stores reads in a vector.

Table 4 compares the implementation overhead of the three predictors. For every predictor, the leftmost two columns correspond to the (rounded) average number of pattern table entries (pte) for every allocated memory block for a history depth of one (d=1) and four (d=4). The third

column indicates overhead (ovh) in terms of the number of bytes with a history depth of one (d=1). All predictors use 4 bits to encode the processor ids. Cosmos uses 3 bits to encode the message type resulting in 7 bits for a history table entry and 14 bits per pte and a total of (7 + 14 pte)/8 bytes per block. MSP and VMSP only use 2 bits to encode the request type, and VMSP uses 16 bits to encode a read vector. Therefore MSP's overhead is (6 + 12 pte)/8 bytes. VMSP requires 18 bits for the history table entry but only 18 + 6 bits for a pte because in VMSP a read vector is always followed by a write or upgrade and a pte will at most contain a single vector. VMSP's overhead is therefore (18 + 24 pte)/8 bytes.

Not surprisingly, the table indicates that MSP and VMSP significantly reduce the number of required pattern table entries as compared to Cosmos. On average, for a history depth of one, Cosmos requires five entries while MSP and VMSP only require three and two entries respectively. MSP reduces the storage requirement in terms of number of bytes to about half of that in Cosmos. Although VMSP uses a less compact encoding, it reduces the overall storage requirement in all but one application as compared to Cosmos. For a history depth of one, however, VMSP significantly improves the prediction accuracy over both MSP and Cosmos and is therefore most cost-effective.

The table also indicates that using a higher history depth to achieve a better accuracy may be impractical. The number of pattern table entries resulting from message reordering becomes prohibitively high in Cosmos specially for *barnes* and *unstructured*.

7.4 Performance of a Speculative DSM

This paper takes a first step in designing and evaluating a speculative coherent DSM using pattern-based predictors. We use a VMSP with a history depth of one as a predictor for our DSMs. Our DSMs primarily rely on two techniques to execute read requests speculatively: (1) Speculative Write-Invalidation (SWI) invalidates writes (or upgrades) early to trigger speculation for a read sequence, and (2) First-Read (FR) uses the arrival of the first read to trigger a read sequence when SWI fails.

Figure 9 illustrates the performance of two speculative coherent DSMs against a Base-DSM system with no speculation. The speculative coherent DSMs are an FR-DSM system only using FR to trigger read sequences and an SWI-DSM using both SWI and FR to trigger read sequences. The graphs plot execution time normalized to those in Base-DSM. The graphs break down execution time into computation time (including barrier synchronization and spinning on locks) and the overall remote request waiting time. The latter indicates an application's potential for performance improvement using a speculative coherent DSM.

The graphs indicate that triggering read sequences using FR alone has the highest impact on execution time and reduces request waiting time in Base-DSM by 10% to 50%. Together SWI and FR reduce request waiting time in four of the applications to 30%-65% of that in Base-DSM. The overall reduction is execution time in SWI-DSM and FR-DSM is on average 12% and 8%, and at best 24% and 17% respectively.

Table 5 depicts the total number of read and write (or upgrades) requests (in thousands) in Base-DSM for all the

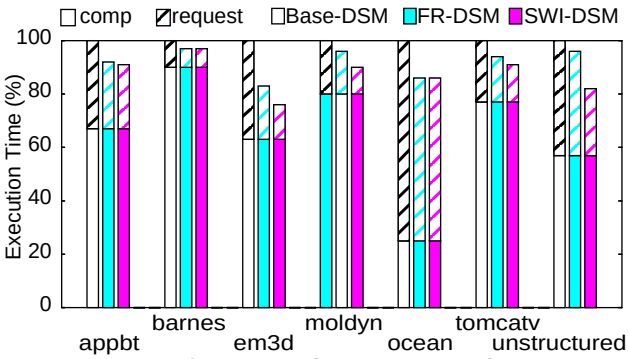

FIGURE 9: Performance improvement in speculative coherent DSMs.

applications. The table also depicts a breakdown of speculation and misspeculation for reads and write invalidates as a percentage of the number requests in FR-DSM and SWI-DSM. *Em3d*, *moldyn*, *tomcatv*, and *unstructured* all benefit from both FR and SWI. *Em3d* exhibits a static producer/consumer sharing pattern. The producer only writes once to a memory block in every iteration and therefore SWI successfully invalidates 98% of the writes and triggers 95% of the reads in SWI-DSM. In FR-DSM, FR can only execute 58% of the reads speculatively because it uses the first read to trigger the rest of the sequence. Overall, SWI-DSM and FR-DSM reduce request waiting time by 70% and 50% as compared to Base-DSM respectively.

Unlike in *em3d*, SWI fails to invalidate successfully all the writes in *moldyn* and *tomcatv*. *Moldyn* exhibits both producer/consumer and migratory sharing. In the producer/consumer phase, the producer reads the blocks shortly after writing to them. As such SWI misspeculates when invalidating the writes. SWI, however, successfully invalidates the writes in the migratory phase accounting for 68% of all the writes (and upgrades) and triggering 40% of the reads. FR captures an additional 39% of all the reads in the producer/consumer sharing phase. The resulting reduction in request waiting time is 50% and 30% for SWI-DSM and FR-DSM as compared to Base-DSM respectively.

Tomcatv is primarily a stencil computation in which processors own and compute sets of rows in matrices and share at the set boundaries. In every iteration, the producers compute and write once in the main phase. However, producers write again to half of boundary blocks in a correction phase before the start of a subsequent iteration. Therefore, SWI only succeeds in invalidating half of the writes and triggering half of the reads reducing request waiting time by a total of 50%. Because the producer first reads then writes, every block has two readers, the producer and the consumer. FR additionally triggers the producer's copy of the read when the consumer's read request arrives and therefore the total of speculative reads adds up to 70% in SWI-DSM. In FR-DSM, all blocks are triggered through FR and only 46% of the reads are speculatively executed reducing request waiting time by only 25%.

Unstructured exhibits a very high degree of read-sharing (i.e., there are on average twelve reads per write or upgrade) in the producer/consumer phase. About half of the reads in the entire application are from this phase. For every read sequence, FR uses one read to trigger a

Application	Base-DSM (x1000)		FR-DSM		SWI-DSM					
			FR read (%)		FR read (%)		SWI read (%)		write invalidate (%)	
	read	write	sent	miss	sent	miss	sent	miss	sent	miss
appbt	832	432	52	14	48	13	10	<1	10	< 1
barnes	1169	458	58	12	52	12	7	1	10	3
em3d	4731	1799	58	0	0	0	95	0	98	0
moldyn	1034	618	39	< 1	39	< 1	40	0	68	< 1
ocean	589	316	41	2	40	2	2	<1	4	2
tomcatv	187	96	46	0	24	0	45	0	48	< 1
unstructured	28461	15985	46	21	23	12	69	10	90	< 1

Table 5: Frequency of requests, speculations, and misspeculations.

sequence of twelve. The other half of the reads are from the reduction phase with migratory sharing patterns. FR can not benefit migratory sharing since the latter only involves read/write pairs. Therefore, FR executes 46% (i.e., eleven out of twelve) of all the reads speculatively in FR-DSM. SWI successfully invalidates 90% of the writable copies of the blocks in *unstructured*. Together with FR, SWI executes 92% of all reads speculatively, and reduces request waiting time by 50% in SWI-DSM as compared to Base-DSM.

SWI does not benefit any of *appbt*, *barnes*, and *ocean* because the simple early-invalidation heuristic for the writes fails in these applications; the producer either reads the block upon writing to it or writes multiple times to the block. These applications all benefit from FR and between 41% and 58% of all reads execute speculatively. *Barnes* exhibits low communication ratios and therefore does not benefit from a reduction in request waiting time.

In *appbt*, much of the request waiting time is in the gaussian elimination phase in which processors proceed in a pipeline and data are passed in a strict producer/consumer manner. FR uses the consumer's read request to execute the producer's read request speculatively. Much of the pipeline's critical path, however, is due to the read request from the consumer and the write/upgrade request from the producer limiting the impact of speculative reads to a reduction of 25% in request waiting time. *Ocean* primarily exhibits near-neighbor sharing. FR uses the consumer's read request to execute the producer's read request (41% of all reads) speculatively. The resulting reduction in request waiting time is about 18%.

The table also indicates that misspeculation frequency is minimal for write invalidates and is only high for speculative reads in applications with low prediction accuracy. The number of write invalidate misspeculations is minimal because MSP prevents further speculative write invalidations upon a misspeculation. Moreover, misspeculated read requests do not impact execution time because the misspeculations primarily result in extra read-only copies of blocks sent to processors which do not actively read the blocks. Because all read-only copies of a block are sent and subsequently invalidated in parallel, the misspeculation penalty is minimal.

8 Conclusions

In this paper, we proposed the Memory Sharing Predictors (MSPs), novel pattern-based predictors—derived from Yeh and Patt's two-level *Pap* branch predictor—to predict and execute coherence protocols speculatively. An MSP is based on the key observation that in order to hide the remote access latency, a predictor must accurately predict only a remote memory access (i.e., a request message) and not the subsequent coherence messages invoked by the access. By eliminating unnecessary coherence messages from the pattern tables, an MSP significantly improves prediction accuracy and implementation cost over previously-proposed general message predictors.

We presented simulation results running shared-memory applications to indicate that: (1) our base MSP eliminates the acknowledgment messages in the pattern tables and increases prediction accuracy in a general message predictor from 81% to 86%, (2) an optimized Vector MSP (VMSP) improves prediction accuracy to 93% by using a compact vector representation of read request sequences and eliminating perturbation due to read request re-ordering, and (3) VMSP not only offers the best prediction accuracy but also reduces implementation storage overhead over a general message predictor.

This paper also took the first step towards designing a speculative coherent DSM using pattern-based predictors. To hide the remote access latency successfully, a predictor must not only predict "what" subsequent memory accesses arrive but also "when" they arrive. We identified simple techniques and mechanisms (that are readily implementable without modifying the base protocol) to trigger and perform speculation for remote read accesses. We presented results from a simple analytic model and detailed simulation of shared-memory applications to indicate that: (1) high-accuracy predictors are the key to high performance in a speculative coherent DSM, (2) triggering read request speculation for a read sequence based on the arrival of the first read reduces execution time in all applications on average by 8% and at best by 17%, and (3) triggering speculation for reads by speculatively invalidating a writable copy reduces execution time on average by 12% and at best by 24%.

9 Acknowledgements

We would like to thank Mark Hill, Alain Kägi, and the anonymous referees for their valuable feedback on earlier drafts of this paper.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991. Also available as U. Washington CS TR 91-03-07.

[5] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[6] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[7] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[8] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[9] Anoop Gupta and Wolf-Dietrich Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[10] Erik Hagersten and Michael Koster. WildFire: A scalable path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[11] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993.

[12] Doug Joseph and Dirk Grunwald. Prefetching Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[13] Stefanos Kaxiras and James R. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 161–170, February 1999.

[14] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[15] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[16] Tom Lovett and Russel Clapp. STiNG: A CC-NUMA compute system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[17] Shubhendu S. Mukherjee and Mark D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[18] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[19] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[20] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[21] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[22] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu Thomas Wicki, and Winfried Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[23] Tse-Yuh Yeh and Yale Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.