To appear in the *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, 1998

Using Prediction to Accelerate Coherence Protocols

Shubhendu S. Mukherjee and Mark D. Hill
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706-1685, USA
{shubu,markhill}@cs.wisc.edu
URL: http://www.cs.wisc.edu/~{shubu,markhill}

Abstract

Most large shared-memory multiprocessors use directory protocols to keep per-processor caches coherent. Some memory references in such systems, however, suffer long latencies for misses to remotely-cached blocks. To ameliorate this latency, researchers have augmented standard coherence protocols with optimizations for specific sharing patterns, such as read-modify-write, producer-consumer, and migratory sharing. This paper seeks to replace these directed solutions with general prediction logic that monitors coherence activity and triggers appropriate coherence actions.

This paper takes the first step toward using general prediction to accelerate coherence protocols by developing and evaluating the Cosmos coherence message predictor. Cosmos predicts the source and type of the next coherence message for a cache block using logic that is an extension of Yeh and Patt's two-level PAp branch predictor. For five scientific applications running on 16 processors, Cosmos has prediction accuracies of 62% to 93%. Cosmos' high prediction accuracy is a result of predictable coherence message signatures that arise from stable sharing patterns of cache blocks.

1 Introduction

Most shared-memory multiprocessors accelerate memory accesses using per-processor caches. Caches are usually made transparent to software with a cache coherence protocol. Most large shared-memory multiprocessors use directory protocols [3, 21, 19]. Directory protocols maintain a directory entry per memory block that records which processor(s) currently cache the block. On a miss, a processor sends a coherence message over an interconnect to a directory, which often forwards message(s) to processor(s) currently caching the block. These processors may forward data or acknowledgments to the requesting processor and/or directory.

Unfortunately, this cache miss and directory activity can disturb a programmer's performance model of shared memory by making some memory accesses tens to hundreds of times slower than others. This problem has led to many proposals, including weaker memory models [2], multithreading [36], non-blocking caches [18], and application-specific coherence protocols [27]. To date, all proposals possess one or more of the following drawbacks: require a more complex programmer interface or model, retard uniprocessor performance, or require sophisticated compilers.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, National Science Foundation with grants MIP-9225097, MIPS-9625558, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

Another class of proposals predict future sharing patterns [7, 13] and take actions to overlap coherence message activity with current work. Predictions can be made by programmers [14, 38, 1, 25], compilers [23, 32, 31, 15], software [4], or hardware. Specialized predictors in hardware include read-modify-write operation prediction in the SGI Origin protocol [19], pair-wise sharing prediction in SCI [34], dynamic self-invalidation [20], and migratory protocols [12, 35]. Other examples of hardware predictors are described in [17, 5, 29, 28]. Existing predictors, however, are directed at specific sharing patterns known *a priori*. Furthermore, a protocol implementation is often made more complex by intertwining predictors with the standard coherence protocol.

This paper seeks a more general predictor to accelerate coherence protocols. Predictors would sit beside each standard directory and cache module to monitor coherence activity and request appropriate actions. If a directory predictor, for example, anticipates that a processor asking for a block B "shared" will subsequently ask for block B "exclusive," the directory can answer the "shared" request with block B "exclusive."

The first contribution of this paper is the design of the *Cosmos* coherence message predictor for accelerating coherence protocols (Section 3). Cosmos' design is inspired by Yeh and Patt's two-level *PAp* branch predictor [39]. Cosmos makes a prediction in two steps. First, it uses a cache block address to index into a *Message History Table* to obtain one or more <processor,message-type> tuples. These <processor,message-type> tuples correspond to sender and message type of the last few coherence messages received for that cache block. Message-type identifies specific coherence actions for a sharing pattern, whereas processor identifies the specific sharers involved in the sharing pattern. Second, Cosmos uses these <processor,message-type> tuples to index a *Pattern History Table* to obtain a <processor,message-type> prediction. Notably, Cosmos faces a greater challenge than branch predictors because the Cosmos' prediction is a multi-bit <processor,message-type> tuple rather than a single bit branch outcome.

This paper concentrates on coherence protocol message prediction in isolation (analogous to studying branch prediction in isolation). We do not integrate the Cosmos predictor into a coherence protocol for two reasons. First, our tools are not ready to handle a full timing simulation of a protocol that can be accelerated using prediction. Second, we do not want initial results in this area obscured by implementation idiosyncrasies. Nevertheless, we expect such integration to be successful because the integration of directed predictions has been successful [19, 20, 12, 35]. Section 4 briefly discusses possibilities for such integration.

The second contribution of this paper is a detailed evaluation of the Cosmos coherence message predictor. Section 5 states methodological assumptions, including the use of five scientific benchmarks on a target shared-memory machine with 16 processors running the Stache directory protocol [30]. Section 6 gives Cosmos' prediction rates and analyzes application details. Variations of Cosmos predict the source and type of the next coherence message with surprisingly-high accuracies of 62-69% (*barnes*), 84-86% (*moldyn*), 84-85% (*appbt*), 74-92% (*unstructured*), and 84-93% (*dsmc*). Cosmos' high prediction accuracy results from predictable coherence message pat-

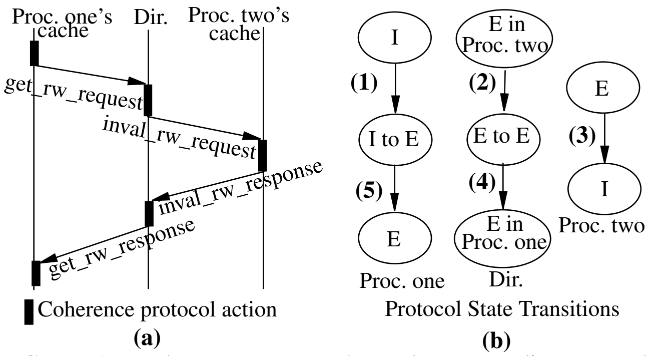


FIGURE 1. (a) shows message exchange between a directory and two caches and (b) shows the corresponding state transitions. Table 1 explains the coherence message types. I = invalid, E = exclusive, Dir. = Directory, Proc. = Processor. Initially, processor two has an exclusive copy of a cache block. Processor one issues a store to the block (1). This invokes processor one's cache coherence protocol, which sends a message to the directory (2). The directory examines its state and sends a message to processor two requesting it to return the block to the directory and invalidate its copy of the block (3). When the directory receives the block from processor two (4), it forwards it to processor one, which marks the cache block as exclusive in processor one (5). The states "I to E" and "E to E" represent transition states.

terns or *signatures* associated with specific cache block addresses. Such signatures are generated by sharing patterns [7, 13] that do not change or change very slowly during the execution of these applications. Cosmos' lower accuracy for *barnes* occurs because *barnes* periodically re-builds its principal data structure (an octree), thereby moving logical nodes (with stable sharing patterns) to different memory addresses (obscuring sharing patterns from Cosmos).

Section 7 explores the implications of Cosmos. Coherence message prediction works because sharing patterns are often stable. Others have exploited sharing patterns with directed optimizations, such as dynamic self-invalidation and migratory protocols. Using Cosmos could be better (or worse) than directed predictors due to performance and implementation issues. Cosmos can perform better because it can discover and track application-specific patterns not known *a priori* (e.g., as occurs for *unstructured*). It can perform worse if it is slower to recognize known patterns. Cosmos' implementation complexity can be less because predictor logic is separated from the standard protocol logic (unlike previous directed predictors that are intertwined with the standard coherence protocol). Cosmos, however, is likely to require more state than directed optimizations. In summary (Section 8), Cosmos' high prediction accuracies justify more investigation into using prediction to accelerate coherence protocols.

2 Background

This section describes the structure of a basic directory protocol (Section 2.1) and reviews Yeh and Patt's two-level adaptive branch predictor (Section 2.2). In the next section we discuss how Cosmos—a modified version of Yeh and Patt's two-level predictor—can predict a directory protocol's messages with high accuracy. Throughout the rest of the paper we will use the terms "node" and "processor" interchangeably because we consider only single-processor nodes to simplify our discussion.

2.1 Structure of a Directory Protocol

Most large-scale shared-memory multiprocessors use a directory protocol to keep multiple caches coherent. A directory protocol associates state with both caches and memory. The state is typically maintained at a cache block (e.g., 32-128 bytes) granularity. The state associated with each memory block is referred to as a directory entry. The directory entry for each memory block records whether or not a memory block is idle (that is, no cached copies exist), a writable copy of the block exists, or one or more readable copies of the block exist.

To simplify our discussion we only consider a full-map and write-invalidate directory protocol, such as the SGI Origin protocol [21]. A directory entry in such a protocol maintains logical pointers to caches that hold a valid copy of the block and invalidates all outstanding copies of the block when one processor wishes to write to it. Similarly, a block in a cache is usually in one of three quiescent states: invalid, shared, or exclusive. These states define whether a processor's load or store can access the cache block. Processors must invoke coherence actions on loads to invalid blocks and on stores to shared (i.e. read-only) and invalid blocks.

A cache coherence protocol can, therefore, be viewed simply as a collection of finite-state machines that change state in response to processor accesses and external messages. For caches, state transitions occur in response to processor accesses and messages from the directory (and possibly other caches). A directory entry changes state in response to messages from caches. Figure 1 shows an example of message exchange and state transitions in two caches and a directory.

Unfortunately, the finite-state machines that implement the coherence logic often incur multiple long-latency operations. These latencies can become severe if coherence actions are implemented in software (e.g., [30]) or firmware (e.g., [22]). Additionally, a directory may need to exchange messages with other caches before it can respond to a processor's request for a memory block. Such message exchange can also introduce substantial delay in the critical path of a remote access. For example, Figure 1a shows that a processor's store to a block that resides in another node's cache may require five coherence protocol actions and four messages. Other protocols differ (e.g., SGI Origin reduce coherence actions to four and messages to three by directly forwarding processor two's response to processor one), but this should have no first-order effect on coherence prediction's usability.

2.2 Two-Level Adaptive Branch Predictor

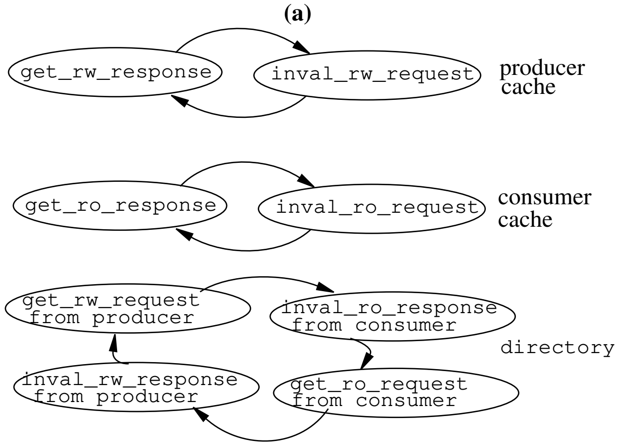
A branch predictor predicts whether the branch will be taken or not taken. Correct prediction of branch directions improves the performance of wide-issue, deeply pipelined microprocessors because it allows them to fetch and execute probable instructions without waiting for the outcome of previous branches. J. Smith [33] proposed several dynamic branch predictors that use program feedback to increase the accuracy of branch prediction. More recently, Yeh and Patt pro-

Messages Received by Directory from Caches		**Messages Received by a Cache from a Directory**	
Message	**Description**	**Message**	**Description**
get_ro_request	get block in read-only (shared) state	get_ro_response	response to get_ro_request
get_rw_request	get block in read-write (exclusive) state	get_rw_response	response to get_rw_request
upgrade_request	upgrade block from read-only to read-write	upgrade_response	response to upgrade_request
inval_ro_response	response to inval_ro_request	inval_ro_request	invalidate read-only (shared) copy of block
inval_rw_response	response to inval_rw_request	inval_rw_request	invalidate read-write (exclusive) copy and return block

TABLE 1. A sample of coherence messages usually found in full-map, write-invalidate coherence protocols.

```
/* private_counter = private variable */
/* shared_counter = shared variable */
repeat
  ...
  if (producer)
    private_counter++
    shared_counter = private_counter
    barrier
  else if (consumer)
    barrier
    private_counter = shared_counter
  else
    barrier
  endif
  ...
until done
```

(a)



(b)

FIGURE 2. This figure shows the incoming message signature generated by a producer-consumer sharing pattern for a cache block. (a) shows a pseudo code for the producer-consumer sharing pattern. A producer writes to a shared counter and a consumer reads the shared counter. (b) shows the sequence of messages received by the producer cache, consumer cache, and directory for the cache block containing the shared counter (assuming no false sharing). Table 1 explains the different message types shown in this figure.

posed a general dynamic branch predictor called *PAp* [39]. PAp is a two-level adaptive predictor that makes a prediction for a branch based on the past behavior of the same branch. First, it uses the program counter of a branch to index into a *Branch History Table* to obtain k bits, which represent the last k outcomes of the branch at this program counter. Second, it uses these k bits to index a Per-Branch *Pattern History Table* to obtain a prediction. Each entry in the Pattern History Table is a finite-state machine, which returns predictions based on the behavior of a finite number of previous occurrences of this branch (and the k bits from the Branch History Table). In the next section we will show how PAp can be modified to obtain coherence message predictions.

3 Predicting Coherence Protocol Messages

This section describes the Cosmos coherence message predictor. The next section briefly outlines how Cosmos can accelerate coherence protocols. This section begins with an example of a producer-consumer sharing pattern and its corresponding coherence message signature. The rest of the section uses this example to describe Cosmos in detail.

3.1 Producer-Consumer Sharing Pattern's Signature

Figure 2 shows an example of a producer-consumer sharing pattern and how it can lead to predictable message patterns or *signatures* for a particular cache block. For example, assuming no false sharing, the producer executing the code in Figure 2a observes the following message sequence for the cache block containing the variable `shared_counter`:

send `get_rw_request` to directory
receive `get_rw_response` from directory
receive `inval_rw_request` from directory
send `inval_rw_response` to directory

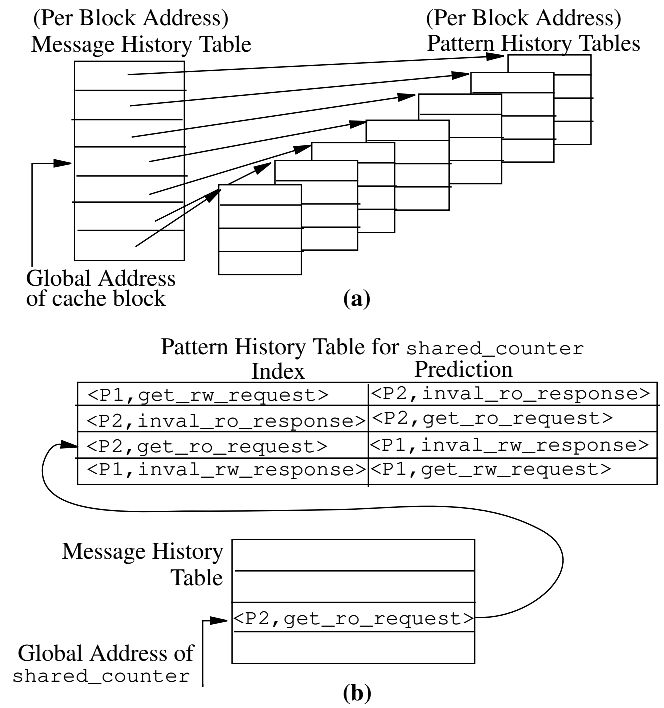


FIGURE 3. (a) shows the logical structure of the Cosmos coherence message predictor and (b) shows an example of how the message and pattern history tables for a directory may look like for the `shared_counter` in Figure 2. In this example, we assume that the last message received by the directory is a get_ro_request from the consumer (denoted as P2). So, Cosmos will predict the next message to be an inval_rw_response from the producer (denoted as P1).

Figure 2b shows the incoming message signature that results from the above message sequence. Note that examining the incoming messages is sufficient to interpret both the sharing pattern (i.e. producer-consumer) and the local processor's actions (i.e., processor store to produced block).

Consider a slightly more complex example in which we extend the pseudo code in Figure 2a to support two consumers instead of one. In this case the producer and the two consumers will still follow the same predictable signatures as shown in Figure 2b. However, at the directory the two `get_ro_request` messages can now arrive in any order from the two consumers. But, the arrival of a `get_ro_request` from the first consumer suggests strongly the possibility of the arrival of another `get_ro_request` from the second consumer and vice versa. To achieve high accuracy a predictor must adapt to such variations in the incoming message stream. The rest of this section discusses the design of such an adaptive predictor called Cosmos.

3.2 Basic Structure of Cosmos

The previous subsection suggests that a coherence message predictor must adapt to an incoming coherence message stream based on two properties:

- address of cache blocks, because sharing patterns of different cache blocks may differ, and
- history of messages for a cache block, because a stream of

incoming coherence messages correspond to fixed sharing patterns for specific cache blocks. [1]

Fortunately, a modified version of Yeh and Patt's two-level adaptive branch predictor called *PAp* [39] satisfies the above requirements! We call such a coherence message predictor *Cosmos*. Given the address of a cache block and the history of messages received for that block, Cosmos can predict with high accuracy the sender and type of the next incoming message for the same block. We allocate a Cosmos predictor for every cache or directory in the machine.

Figure 3a shows the logical structure of Cosmos. Cosmos is a two-level adaptive predictor. The first-level table—called the *Message History Table (MHT)*—consists of a series of *Message History Registers (MHRs)*. Each MHR corresponds to a different cache block address. An MHR contains a sequence of `<sender, type>` tuples corresponding to the last few coherence messages that arrived at the node for the specific cache block. We call the number of tuples maintained in each MHR the *depth* of the MHR.

The second-level table of Cosmos consists of a sequence of *Pattern History Tables (PHT)*, one for each MHR. Each PHT contains prediction tuples corresponding to possible MHR entries. Each PHT is indexed by the entry in the MHR entry. The next two subsections outline how to obtain predictions from and update entries in Cosmos.

Figure 3b shows the entries in an MHR and its PHT corresponding to the `shared_counter` variable in Figure 2. The MHT in Figure 3b has a depth of one, so this MHR entry contains only one `<sender, type>` tuple. The `<P2, get_ro_request>` tuple shown in this figure denotes that the last message received for the cache block containing the `shared_counter` is a `get_ro_request` message from the processor P2, which is consumer of the `shared_counter` in this case. The corresponding PHT captures patterns of messages received for `shared_counter`. For example, earlier Cosmos observed a `get_ro_request` message from processor P1 followed by an `inval_ro_response` from processor P2. The first entry of the PHT reflects this relationship. Thus, Cosmos will predict the arrival of an `inval_ro_response` message from processor P2, next time it sees a message `get_ro_request` from processor P1. Because the MHR contains the tuple corresponding to the last message received, to obtain a prediction we simply find the correct MHR, and use that entry to index into the PHT, which will give us a prediction if an entry exists for that tuple.

Cosmos differs from Yeh and Patt's two-level adaptive branch predictor called PAp (see Section 2.2) in three ways. First, the first-level table in Cosmos is indexed by the address of a cache block, whereas PAp is indexed by the program counter of a branch. Second, Cosmos must choose one prediction from several alternatives, whereas PAp usually chooses between two alternatives—branch taken or branch not taken. Third, the state machine in each PHT entry in PAp encodes the history of the last few outcomes of the same branch. Instead, a PHT entry in Cosmos simply consists of a prediction. Additionally, PHT entries in Cosmos can contain state machines (Section 3.6), but these are typically used as filters to remove noise from the incoming message stream.

Below we outline the exact steps involved in obtaining a prediction from and updating Cosmos. Specific implementations of Cosmos may either separate or combine these two steps.

1. Cosmos could predict the next coherence protocol state, instead of the next incoming coherence message. We believe these two approaches are equivalent. However, Cosmos predictors for specific protocols may consume less space if Cosmos captures messages, instead of coherence protocol states. For example, at the directory the coherence protocol state of the Wisconsin Stache protocol (Section 5.1) consumes eight bytes, whereas the message information could be captured in two bytes (Table 7).

3.3 Obtaining Predictions from Cosmos

Here are the steps involved to obtain a prediction from Cosmos:

- index into the MHR table with address of a cache block,
- use the entry in MHR to index into the corresponding PHT, and
- return the prediction entry (if one exists) in the PHT as the predicted tuple, which contains the predicted sender and type of the next incomng message corresponding to that cache block; otherwise, return no prediction.

3.4 Updating Cosmos

Typically, we expect Cosmos to be updated after every message reception when we know for sure the `<sender, type>` tuple of a message. Here are the steps involved in updating Cosmos:

- index into the MHR table with the address of a cache block,
- use the entry in MHR to index into the corresponding PHT,
- write new `<sender,type>` tuple as new prediction for the index corresponding to the MHR entry, and
- left shift the `<sender,type>` tuple into the MHR for the cache block.

3.5 How Cosmos Adapts to Complex Signatures?

Cosmos can adapt to complex message streams, such as the one outlined at the end of Section 3.1. If two `get_ro_request` messages arrive out of order from two different consumers (P1 and P2), the PHT table will contain the following two entries:

Index	**Prediction**
`<P1, get_ro_request>`	`<P2, get_ro_request>`
`<P2, get_ro_request>`	`<P1, get_ro_request>`

Therefore, Cosmos can effectively predict the next incoming coherence message, even though incoming messages may arrive in a different order in different instances. [2]

For more complicated sequences of incoming messages, Cosmos may need an MHR with depth greater than one. For example, if three `get_ro_request` messages come out of order from three consumers (P1, P2, and P3), then the PHT for a Cosmos predictor with MHR of depth = 2 may contain the following three entries:

Index	**Prediction**
`<P1, get_ro_request>, <P2, get_ro_request>`	`<P3, get_ro_request>`
`<P2, get_ro_request>, <P3, get_ro_request>`	`<P1, get_ro_request>`
`<P3, get_ro_request>, <P1, get_ro_request>`	`<P2, get_ro_request>`

Clearly, this allows Cosmos to predict the third incoming coherence message accurately based on the history of previous messages. Fortunately, several studies (e.g., [13, 38, 25]) have shown that the average number of sharers of a cache block is usually less than two. Consequently, we do not expect the depth of the MHR to be very high for most applications. Specifically, we found that an MHR of depth three is sufficient in most cases for the five parallel applications we studied in this paper. [3]

2. A more aggressive predictor could ignore the senders for the `get_ro_request` messages. However, this may not be possible if there are intervening messages of other types for the same cache block.

3. We cannot ignore the processor number of the messages, even though it may appear so in the example. This is because actions taken by the directory (e.g., sending an invalidation message) based on the prediction may require accurate prediction of the processor number. However, it may be possible to group the processor numbers into a set and perform actions on the entire set of processors.

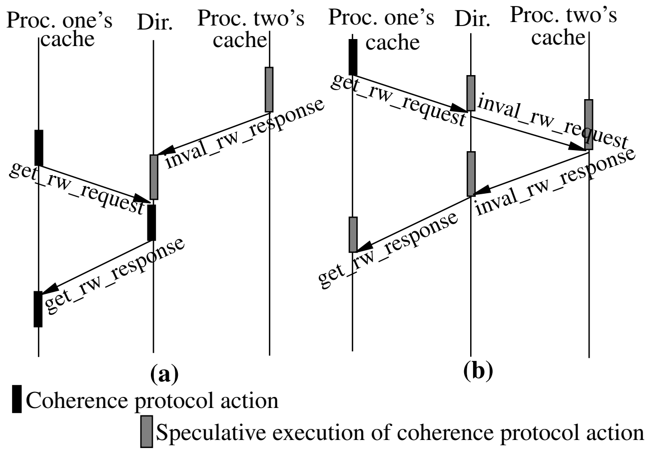


FIGURE 4. Two examples of using prediction to accelerate coherence protocols. (a) shows a protocol in which protocol actions are accelerated in anticipation of processor one's write miss. (b) shows a protocol that predicts incoming coherence messages, updates protocol state, generates (but does not send) messages speculatively, and commits protocol state and messages only if the predicted message arrives. Dir. = Directory, Proc. = Processor. Section 4 outlines possible triggers for the speculative actions shown in this figure.

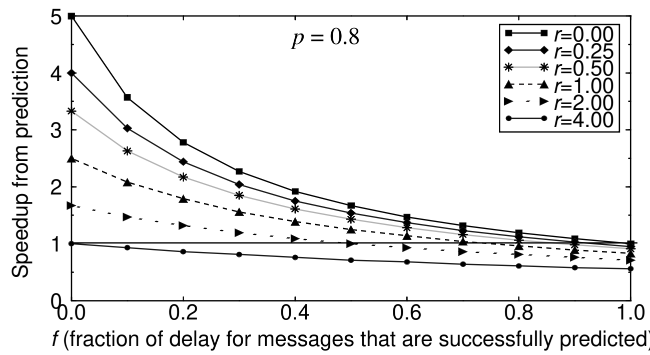


FIGURE 5. This figure displays a crude execution model that translates coherence message prediction rates into a parallel program's speedup. We assume that execution time is determined purely by the delay of messages in the critical path of the program. Results in this figure (based on a prediction rate of 0.8 for all graphs) show that coherence prediction can result in substantial speedups.

3.6 Filtering Noise from Coherence Message Stream

When updating Cosmos we can use *filters* to reduce noise from the coherence message stream in the same way Yeh and Patt's PAp predictor removes noise from a stream of branches. For example, if 99% of the time, message B follows message A, then on seeing message A, Cosmos will predict the next message to be B. We do not want our prediction to change if these messages arrive rarely in the sequence: A, C, and B, instead of the sequence A, B. Branch predictors have a similar problem when programs exit loops. Frequently, the exit from loops is a taken branch; however, when the loop is executed completely, the exit is a not-taken branch. Branch predictors typically avoid updating their prediction on exiting a loop via a two-bit saturating counter proposed by J. Smith [33]. One bit of the two-bit counter represents the direction of the branch and other bit represents the counter. Because a message needs more than one bit to represent a `<sender, type>` tuple, we simplify the counter and use only a single bit. With this single-bit counter, we update the prediction for a cache block to a different message only if we see two consecutive message mis-predictions for the same block.

Our results (Section 6.2) suggest that filters increase the prediction accuracy for Cosmos predictor with MHR depth of one, but they do not help Cosmos predictors with MHR depth greater than one. This is because both history and filters reduce noise from the message stream. However, history information adapts to the noise, while filters simply remove it.

3.7 Implementation Issues for Cosmos

Cosmos is a two-level adaptive predictor with the first level containing message history registers (MHRs) and the second level containing pattern history tables (PHT). It may be possible to merge the first-level table with the cache block state maintained at both directories and caches. However, this may lead to a loss of Cosmos' history information when cache blocks are replaced. This problem may not arise for the directory because directory state is usually persistent during the entire duration of a parallel application.

The second-level table is more challenging to implement because it may require large amounts of memory to capture pattern histories for each cache block. However, our results (Section 6.2) show that Cosmos' memory overhead for 128 byte cache blocks is less than 14% for an MHR depth of one. This is because the number of pattern histories corresponding to a cache block is low, that is, less than four (on average) for an MHR depth of one for all five applications we studied in this paper. Consequently, we could preallocate four pattern history entries corresponding to each cache block. If a cache block needs more pattern histories, then it can allocate them from a common pool of dynamically allocated memory in the same way LimitLESS [10] directory entries capture the list of sharers for a particular cache block. Nevertheless, higher prediction accuracies may require greater MHR depths, which may result in larger amounts of memory.

4 Using Coherence Protocol Message Predictors

This section briefly discusses how a coherence protocol message predictor, such as Cosmos, can be integrated with a coherence protocol. Predictors would sit beside each standard directory and cache module and accelerate coherence activity in two steps. First, they would monitor message activity and make a *prediction.* Second, based on the prediction, they will invoke an *action* in the standard coherence protocol. Key challenges include mapping predictions to actions, performing actions at the right time (not too early or late), dealing with mis-predictions, and determining how coherence prediction affects runtime.

4.1 Mapping Predictions to Actions

Mapping predictions to actions is straightforward in many cases. Table 2 lists several examples of prediction-action pairs. For example, a directory action corresponding to a read-modify-write prediction for a block would be to return the block to the requesting cache in "exclusive" state, instead of the "shared" state.[1] Figure 4a shows another example where the predictor in processor two's cache predicts a write miss from another processor. A consequent action—as done by an implementation of Lebeck and Wood's dynamic self-invalidation protocol [20]—would be to replace the block from processor two's cache to the directory before the directory receives the write miss request from processor one's cache. More generally, each directory and cache can predict incoming coherence messages, execute protocol actions speculatively (which may include sending messages speculatively), and take appropriate actions on mis-predictions (Figure 4b). Speculative execution of coherence protocol action may also involve executing a sequence of protocol actions, instead of executing a single action (that is normally done). This allows a directory and a cache to optimize for sharing patterns not known *a priori*.

1. Cosmos can identify a read-modify-write operation from the signature: `<P,get_ro_request>, <P,upgrade_request>`.

Prediction	Prediction Location	Static/ Dynamic	Action	Protocol
Load/store from processor	Cache	Static	Prefetch block in shared or exclusive state	Stanford DASH protocol [21]
Read-modify-write	Directory	Static	Directory responds with block in exclusive state on read miss for idle block	SGI Origin protocol [19]
Read-modify-write	Cache	Static	Cache requests exclusive copy on read miss	Dir_1SW [14], Dir_1SW+ [38]
Store from different processor	Cache	Static	Replace block and return to directory	Dir_1SW [14], Dir_1SW+ [38]
Store from different processor	Directory	Dynamic	Invalidate and replace block to directory if exclusive	Dynamic Self-Invalidation [20]
Block migrates between different processors	Directory	Dynamic	On read miss return block to requesting processor in exclusive state	Migratory protocols [12, 35]

TABLE 2. Examples of prediction-action pairs in existing protocols

4.2 Determining When to Perform Actions

Detecting when to perform actions is simple in some cases, but can be tricky in others. An obvious time to trigger actions would be to do so on certain protocol transitions. For example, the directory can trigger the action corresponding to a read-modify-write prediction when a read miss request arrives for a block. In Figure 4a, processor two's cache can trigger the block replacement action when it sees `inval_rw_request` messages for other spatially contiguous blocks.[1]

4.3 Detecting and Handling Mis-predictions

Directories and caches can detect prediction success or failure by simply verifying whether the next message for a cache block is indeed the predicted message or not. Additionally, if any action sends messages speculatively to other directories or caches, then these directories or caches must be informed of the mis-prediction. This allows a directory or a cache to recover from mis-predictions caused by other directories and caches.

Mis-predictions can leave the processor state, protocol state, or both in an inconsistent state. Consequently, a protocol must recover from mis-predictions. In general, actions can be classified into three categories. Below we outline possible recovery mechanisms for each action.

- Actions that move the protocol between two "legal" states require no recovery on mis-prediction. Replacement of a cache block that moves the block from "exclusive" to "invalid" state is an example of such an action (Figure 4a). While there is no explicit recovery in this example, a mis-prediction may still hurt performance by resulting in an extra cache miss for the replaced block.
- Actions that move the protocol state to a future state, but do not expose this state to the processor, can recover from mis-predictions transparently. On detecting a mis-prediction a protocol simply discards the future state. On detecting a prediction success, however, the coherence protocol state must commit the future state and expose it to the processor. Mis-predictions corresponding to actions in Figure 4b can use such recovery actions.
- Actions that allow both the processor and the protocol to move to future states need greater support for recovering from mis-predictions. Before speculation begins both the processor and the protocol can checkpoint their states. Then, on detecting a mis-prediction, both the processor and the coherence protocol must roll back to the checkpointed states. On detecting a success, the current protocol and processor states must be committed. Such actions can be created by coupling a speculative processor, such as the MIPS R10000, with a coherence protocol accelerated with prediction.

1. Note that the activity of a processor can be interpreted from the incoming coherence message, even though a Cosmos predictor sitting next to a processor cache predicts incoming coherence messages (and not local processor activity). For example, a Cosmos prediction of `get_ro_response` suggests that the local processor will incur a read miss for a cache block. This information can be used to properly "time" an action corresponding to a prediction.

4.4 How Coherence Prediction Affects Performance?

A final aspect of coherence prediction is determining exactly how it affects application runtime. As stated in the introduction, this paper concentrates primarily on prediction accuracies. Nevertheless, it would be useful to gain a rough understanding of how prediction affects runtime.

A simplistic execution model is as follows. Let:

- p be the prediction accuracy for each message,
- f be the fraction of delay incurred on messages predicted correctly (e.g., f=0 means that the time of a message predicted correctly is completely overlapped with other delays), and
- r be the penalty due to a mis-predicted message (e.g., r=0.5 implies a mis-predicted message takes 1.5 times the delay of a message without prediction).

If performance is completely determined by the number of messages in the critical path of a parallel program, then speedup due to prediction is:

$$\frac{time\ (without\ prediction)}{time\ (with\ prediction)} = \frac{1}{p * f + (1 - p) * (1 + r)}$$

Figure 5 displays the model's result for a prediction accuracy of 80% (p=0.8) The model shows, for example, that speedup can be as high as 56% with a mis-prediction penalty of 100% (r=1) and a prediction success benefit of 30% (f=0.3).

5 Methodology

We evaluate Cosmos' prediction accuracy using traces of coherence messages obtained from the Wisconsin Stache protocol (Section 5.1) running five parallel scientific applications (Section 5.2). Each application has a start-up phase to initiate the computation (e.g., initiate data structures). Our traces do not contain coherence messages generated in this start-up phase. The traces were generated by the Wisconsin Wind Tunnel II simulator [26] simulating a 16-node parallel machine, with each node having one processor, a coherent memory bus, and an optimized network interface [24].

The simulated parameters are shown in Table 3. Cosmos' prediction accuracy is largely insensitive to variations in network latency. For example, changing the network latency from 40 nanoseconds (Table 3) to one microsecond hardly changes Cosmos' prediction rates of the five applications we studied in this paper.

5.1 Wisconsin Stache Protocol

We obtained our coherence message traces from the Wisconsin Stache protocol. Stache is a software, full-map, and write-invalidate directory protocol that uses part of local memory as a cache for remote data [30].

Table 1 shows all the types of coherence messages generated by

Number of parallel machine nodes	16
Processor speed	1 GHz
Cache block size	64 bytes
Cache size	one megabyte
Cache associativity	direct-mapped
Main memory access time	120 ns
Memory bus coherence protocol	MOESI
Memory bus width	256 bits
Memory bus clock time	250 MHz
Network message size	256 bytes
Network latency	40 ns
Network Interface access time	60 ns

TABLE 3. System Parameters

Stache. These coherence messages are common to most full-map, write-invalidate directory protocols. For all our Stache simulations we use a (software) cache block size of 128 bytes.

Our implementation of Stache differs other full-map, write-invalidate coherence protocols in five ways:

- Unlike the DASH protocol, Stache uses the *half-migratory optimization*. In this optimization a directory requests a cache to mark an exclusive block invalid, and not shared, when it receives a read or write miss request from another cache. This is beneficial if this same cache block is not immediately read from the former cache.
- Our Stache implementation allocates pages in round-robin fashion across the 16 nodes. For example, if page X is allocated to node 10, then page X+1 will be allocated to node 11. The owner of each page functions as the directory for that page. The directory pages are optimized to function as cache pages for the local node. Consequently, in most cases Stache does not generate local messages between the cache and directory within a particular node.
- Cache blocks on a cache page in a local node communicate only with one specific directory page in another node. Consequently, for blocks on a cache page, the sender is always a fixed node containing the directory page. A directory page can, however, receive messages from any node caching the page.
- Currently, Stache does not replace pages (and, hence, cache blocks) from the portion of local memory it designates as a cache for remote memory. This implies that Cosmos' history information for cache blocks persists over time. Protocols that replace cache blocks may need to preserve the history information even after the block is replaced. Alternatively, such protocols can speculate only at the directory, where Cosmos' history information is persistent during the duration of a parallel application.
- Barriers are implemented with point-to-point messages. Consequently, our prediction accuracies do not include prediction rates for barrier variables.

Benchmarks	**Input Data Sets**	**Iter**
appbt	24x24x24 cubes	30
barnes	16K particles	19
dsmc	48600 initial particles, 9720 cell	320
moldyn	2048 particles	40
unstructured	9428 nodes, 59863 edges, 5864 faces	10

TABLE 4. Benchmarks. Iter = number of iterations. *appbt* is from NASA Ames [6] and parallelized at the University of Wisconsin [9], *barnes* is from the Stanford SPLASH-2 suite [37], and *dsmc*, *moldyn*, and *unstructured* are from the Universities of Maryland and Wisconsin [27].

Nevertheless, Cosmos' prediction results with Stache should not be significantly different from what would be obtained with a full-map, write-invalidate directory protocol.

5.2 Benchmarks

Table 4 depicts the five benchmarks used in this study. *Appbt* is a parallel three-dimensional computational fluid dynamics application [9] from the NAS benchmark suite. The code is spatially parallelized in three dimensions. The main data structures are a number of 3D arrays, each of which is divided up among different processors as 3D sub-blocks. Each processor is responsible for updating the sub-block it owns. Sharing occurs between neighboring processors in 3D along the boundaries of these sub-blocks.

Barnes simulates the interaction of a system of bodies in three dimensions using the Barnes-Hut hierarchical N-body method [37]. The main data structure is an octree. The octree's leaves contain information about each body and internal nodes represent space cells. In each iteration the octree is rebuilt and traversed once per body to compute the forces on individual bodies. The communication pattern induced by such traversals is quite irregular.

Dsmc studies the properties of a gas by simulating the movement and collision of a large number of particles in a three-dimensional domain with discrete simulation Monte Carlo method [27]. *Dsmc* divides domains into cells in a static Cartesian grid. Each cell contains particles, which collide only with other particles in the cell. The cells are spatially divided up among processors. At the end of each iteration, particles move from one cell to another. The primary communication occurs during this movement.

Moldyn is a molecular dynamics application, whose computational structure resembles the non-bonded force calculation in CHARMM [8]. Molecules in *moldyn* are uniformly distributed over a cuboidal region with a Maxwellian distribution of initial velocities. A molecule's velocity and force exerted by other particles determine the molecule's position. Force computation limits interactions to molecules within a cut-off radius. An interaction list—rebuilt every 20 iterations—records pairs of interacting molecules. The arrays that record the force exerted on molecules and molecules' coordinates induce the

Depth of MHR	***appbt***			***barnes***			***dsmc***			***moldyn***			***unstructured***		
	C	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**
1	91	77	**84**	80	42	**62**	94	73	**84**	92	79	**86**	85	65	**74**
2	90	79	**85**	81	56	**69**	95	77	**86**	91	80	**86**	90	86	**88**
3	89	80	**85**	79	57	**69**	94	92	**93**	90	79	**85**	90	88	**89**
4	89	80	**85**	78	56	**68**	94	92	**93**	90	77	**84**	96	88	**92**

TABLE 5. Prediction rates (expressed in percentage of hits) obtained from Cosmos. Depth of MHR denotes the number of messages used by Cosmos to predict the next incoming coherence message. C = prediction rate at cache, D = prediction rate at directory, and O = overall prediction rate.

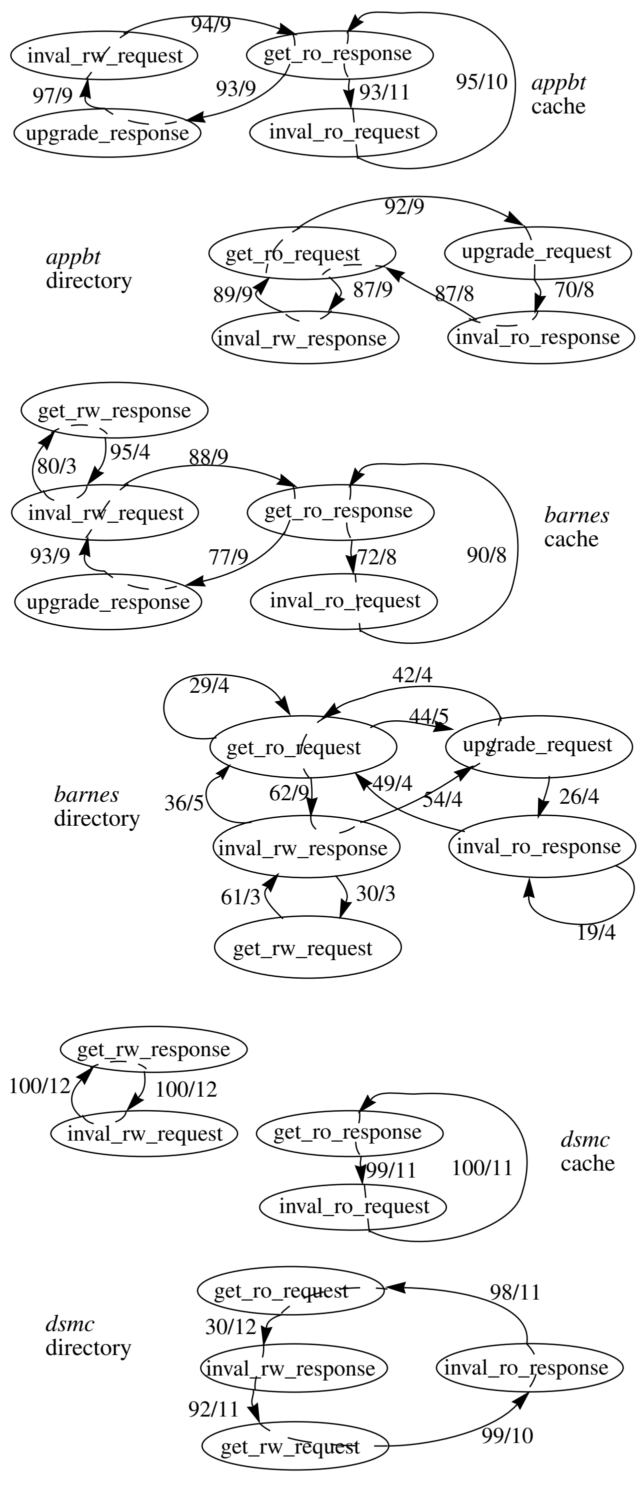


FIGURE 6. Dominant (incoming) message signatures for *appbt*, *barnes*, and *dsmc* at the cache and directory. Arcs represent the order in which two messages arrived. Each arc is labelled as X/Y, where X = percentage of correct predictions for that particular arc and Y = percentage of references to that arc. For example, an arc labelled 94/9 is predicted correctly 94% of the time and constitutes 9% of the total references to all arcs. All X and Y numbers are measured with a Cosmos predictor with MHR depth of one. All Y for a benchmark do not add upto 100% because we only present the dominant transitions we observe. The dotted lines represent dominant message signatures observed in the message stream.

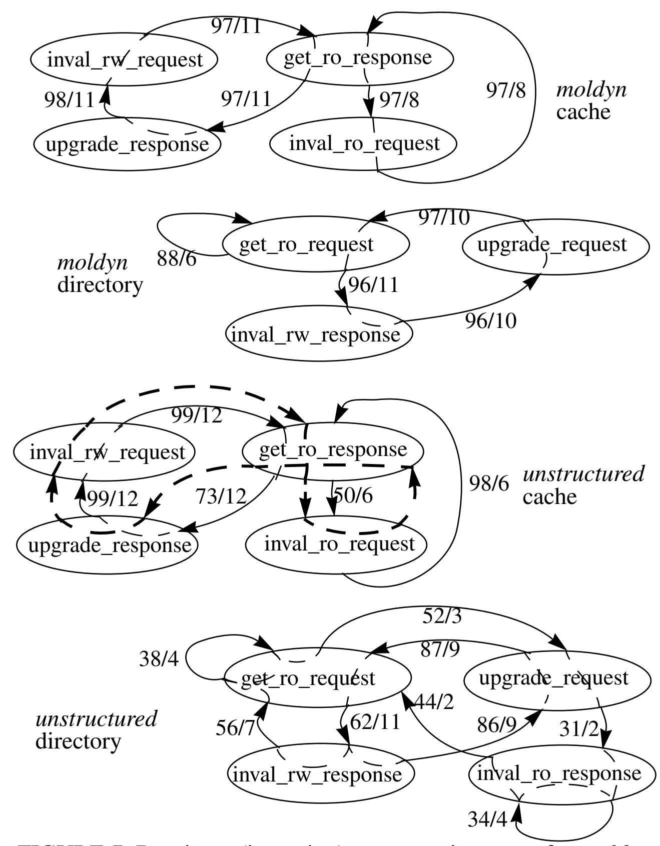


FIGURE 7. Dominant (incoming) message signatures for *moldyn* and *unstructured*. See caption of Figure 6 for an explanation of the figure. We show *unstructured*'s second dominant message signature (at the cache) using bold and dashed lines.

maximum communication.

Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body [27]. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges, and faces. Most communication occurs along the edges and faces of the mesh.

6 Results

In this section we examine Cosmos' basic prediction accuracy (Section 6.1) and then delve into Cosmos' sensitivity to noise and initialization effects and Cosmos' memory requirements (Section 6.2).

6.1 Basic Prediction Rate

Table 5 shows that Cosmos achieves high prediction accuracy. With an MHR depth of one, Cosmos' overall prediction accuracy ranges between 62-86%. Cosmos achieves such high accuracy because cache blocks in most applications generate predictable coherence message signatures. These signatures are related directly to sharing patterns of an application's data structures. All our applications, except *barnes*, have fixed signatures (see Figures 6 and 7) throughout the entire execution of the parallel application. *Barnes* has slightly lower accuracy because shared-memory addresses are reassigned to different objects across iterations. Below we discuss each application's prediction accuracy in detail.

Table 5 shows that Cosmos has higher accuracy for a cache compared

Depth of MHR	***appbt***			***barnes***			***dsmc***			***moldyn***			***unstructured***		
	0	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**
1	84	85	85	62	66	66	84	86	86	86	86	86	74	78	78
2	85	85	86	69	71	71	86	88	88	86	86	86	88	89	89

TABLE 6. This table shows the prediction accuracy of Cosmos as we vary the maximum count of the saturating counter from 0 to 2. The saturating counter filters noise from the coherence message stream (Section 3.6). The overall prediction rates in Table 5 correspond to this table's column 0 (i.e. no filter).

to a directory. For the Stache protocol, a cache receives messages from a fixed sender—that is, a fixed directory, which limits the number of `<sender,message-type>` tuples Cosmos must choose its predictions from. In contrast, a directory receives messages from multiple caches (i.e. senders) for the same cache block. Consequently, Cosmos' predictions are more accurate for Stache caches than Stache directories.

Table 5 also shows that Cosmos' prediction accuracy usually increases with the increase in the MHR depth. With MHR depth of two, the accuracy ranges between 69-88%, while a depth of three results in prediction accuracy that ranges between 69-93%. Having history information helps because it allows Cosmos to recognize predictable coherence streams (Section 3.5). However, most of our applications do not benefit beyond an MHR depth of three (Table 5).

Below we examine why Cosmos achieves high prediction rates for each of the five applications. Surprisingly, variations in simple sharing patterns studied by Bennett, et al. [7] and Weber and Gupta[13], can lead to sequences of coherence actions (and consequent signatures) that are significantly different from those generated by simple sharing patterns (e.g., see *unstructured*'s sequence of messages below). Consequently, predictors based on simple sharing patterns may not be able to correctly speculate the sequence of coherence actions that may be generated. However, Cosmos can capture such variations in sharing patterns because Cosmos adapts to the incoming message stream, which directly determines the sequence of coherence actions to follow.

***Appbt*'s** high prediction accuracy results from its producer-consumer sharing pattern. *Appbt* is a three-dimensional stencil-style code in which a cube is divided up into subcubes. Each subcube is assigned to one processor. Communication occurs between neighboring processors along boundaries of the subcubes.

The sharing pattern that results in the sequence of messages shown for *appbt* in Figure 6 is: producer reads, producer writes, and consumer reads. This pattern repeats for most cache blocks throughout the entire application. Consequently, Cosmos adapts well to *appbt* resulting in a prediction accuracy of 85%.

Note that the half-migratory optimization discussed in Section 5 hurts appbt because the producer first reads a block before writing to it. In the absence of this optimization, the producer pattern would have simply cycled through the two messages: `inval_rw_request` and `upgrade_response`. Clearly, a dynamic predictor, such as Cosmos, can be used to either inhibit (e.g., *appbt*) or trigger (e.g., see discussions on *dsmc* and *moldyn* below) the half-migratory optimization.

Figure 6 shows that all transitions for *appbt* have high prediction accuracy except the transition from `upgrade_request` to `inval_ro_response` at the directory. The low accuracy on this transition results from false sharing in two data structures. It appears that this false sharing generates multiple signatures that the protocol oscillates between randomly. This confuses the predictor resulting in lower accuracy.

***Barnes*'s** prediction accuracy ranges between 62-69% for different MHR depths. This is slightly lower than that for our other applications. This is because in *barnes* nodes of the octree can be assigned different shared-memory addresses in different iterations. Unfortunately, Cosmos cannot make accurate predictions for these nodes of the octree. This is because Cosmos' prediction is based on information it collected on past behavior (e.g., previous iterations) of a particular shared-memory address (at a cache block granularity).

Figure 6 shows that *barnes* has a variety of sharing patterns, some of which exhibit dominant signatures throughout the execution of the program. However, the low accuracies on most arcs improve with more history information (i.e. greater MHR depth).

Dsmc shows the highest accuracy among all our applications. *Dsmc*'s dominant sharing pattern is the classical producer-consumer pattern in which the producer writes and the consumer reads shared cache blocks. This happens because at the end of each iteration *dsmc* communicates information between two processors via shared buffers. This leads to the message sequence shown in Figure 6. Note that the half-migratory optimization helps *dsmc* because the producer does not read the data before it writes to it. Consequently, invalidating the producer's cache blocks, instead of converting them to read-only, avoids an extra handshake with the directory.

Figure 6 shows that the transition from `get_ro_request` to `inval_rw_response` has a low prediction accuracy. This low accuracy, however, disappears with increased MHR depth because updates to shared buffers frequently follow deterministic patterns. Nevertheless, in some cases multiple processors compete for exclusive access to a shared buffer. This creates somewhat oscillating patterns that confuse Cosmos. Fortunately, Cosmos learns to isolate these cases using either more history information or via noise filters (see Section 6.2).

***Moldyn*'s** high accuracy results from two dominant sharing patterns: migratory and producer-consumer patterns. The migratory sharing pattern results in the message sequence `<get_ro_response, upgrade_response, inval_rw_response>` in both processors a block is migrating between. The same pattern is exhibited for the producer in the producer-consumer pattern. However, the consumer for the producer-consumer pattern sees the sequence: `<get_ro_response, inval_ro_request>`. Hence, the number of references to the pattern `<get_ro_response, upgrade_response, inval_rw_response>` is greater than the number of references to the pattern `<get_ro_response, inval_ro_request>` (Figure 7). The sequence seen at the directory results primarily from the migratory pattern.

Moldyn's migratory pattern results from the way it reduces a shared array, which contains force calculations for simulated molecules. In each iteration each processor collects its contribution for different elements of the shared array in a private array. At the end of the iteration each processor adds its contribution from the private array to the shared array. Updates to each element in the shared array happens in a critical section, which results in the migratory pattern. Note that the half-migratory optimization helps *moldyn*. In the absence of this optimization *moldyn* would have probably seen the signature: `<get_ro_request, inval_rw_response, upgrade_request, inval_ro_response>` at the directory.

Moldyn's producer-consumer sharing pattern results from updates to a shared array that contains the coordinates of simulated molecules.

Depth of MHR	*appbt*		*barnes*		*dsmc*		*moldyn*		*unstructured*	
	Ratio	Ovhd	Ratio	Ovhd	Ratio	Ovhd	Ratio	Ovhd	Ratio	Ovhd
1	1.2	5.4%	3.8	13.5%	0.8	3.9%	0.8	4.0%	1.7	6.8%
2	1.4	9.6%	6.9	35.4%	0.4	5.1%	1.1	8.3%	2.1	12.8%
3	1.9	16.4%	9.3	63.0%	0.3	6.7%	1.6	14.9%	2.8	21.9%
4	2.6	26.5%	10.9	91.8%	0.3	8.9%	2.0	21.6%	3.4	33.0%

TABLE 7. Memory overhead of Cosmos predictors (with no filter). Ratio = total number of PHT entries / total number of MHR entries. MHR entries correspond to cache blocks that were referenced at least once in the parallel section of an application. Ovhd expresses the average memory overhead per 128-byte block as a percentage of the block size. More precisely, Ovhd = (tuple size * [MHR depth + Ratio * (MHR depth + 1)] * 100 / 128)%. We assume the tuple size of two bytes (12 bits for processors and 4 bits for coherence message types). Note that some Ratios are less than one. This is because unless the number of protocol references to a cache block is greater than the MHR depth, we do not allocate a PHT for that MHR. This makes all of *dsmc*'s Ratios less than one because some of *dsmc*'s shared-memory data structures are touched rarely. For the same reason, unlike other benchmarks, *dsmc*'s Ratio decreases with increase in MHR depth because the number of such shared-memory blocks that are touched greater number of times than the MHR depth is even fewer.

Moldyn's producer-consumer pattern results in message signatures similar to that of *appbt*'s at both the producer and consumer caches. However, the average number of consumers for *moldyn* is 4.9, whereas for *appbt* the number of consumers is one. Consequently, at the directory we see back-to-back `get_ro_request` messages arriving with high predictability.

Unstructured is different from the rest of our applications because it has different dominant signatures for the same data structures in different phases of the application. The same data structures oscillate between migratory and producer-consumer sharing patterns. The migratory sharing pattern is similar to *moldyn*'s and occurs when each processor updates different elements of the shared arrays in critical sections. The migratory pattern is followed by the producer-consumer pattern in which a producer is itself a consumer of the data. The average number of consumers per producer is 2.6. The signature shown in bold and dashed arrows in Figure 7 represents the transition from migratory to producer-consumer pattern. The directory sees corresponding signatures.

Figure 7 shows that *unstructured*'s prediction accuracy for several arcs with MHR depth of one is low. This is because of the change in sharing pattern. Table 5 shows, however, that Cosmos' accuracy increases from 74% to 92% as the MHR depth increases from one to four. This increase in prediction accuracy from the increase in MHR depth also results in high prediction accuracies for these arcs.

6.2 Additional Analysis

Effect of Filters on Prediction Accuracy. Noise filters can increase the prediction accuracy of Cosmos. We implement Cosmos' noise filter as a saturating counter, which counts upwards from zero and saturates at a maximum count. Table 6 shows the prediction accuracy of Cosmos as we vary the maximum count between 0 and 2.

Filters increase prediction accuracy slightly (up to around 6%) only for Cosmos predictors with MHR depth of one. For MHR depth of two or beyond filters do not help much. This is because both filters and history information remove noise from the message stream. However, history information allows Cosmos to learn from and adapt to the noise. Consequently, if the noise repeats, then Cosmos can achieve higher accuracy. In contrast, filters simply remove noise, but do not let Cosmos adapt to it. Hence, predictors with filters and MHR depth of one achieve lower accuracy than predictors with greater MHR depths. Additionally, filters do not help predictors that have MHR depth greater than one.

Time to Adapt. A critical question for predictors, such as Cosmos, is how long it takes them to achieve the steady-state prediction rates. Cosmos predictors need time to achieve steady-state behavior because they adapt to the incoming stream. We use number of iterations of each application as an approximation to time. This is because the five parallel applications we examined in this paper iterate over a number of steps or iterations. Cosmos can predict incoming coherence messages for a cache block fairly accurately because sharing pattern of a cache block in one iteration is usually similar to its sharing pattern in the previous iteration.

We found that *unstructured* and *barnes* achieve steady-state behavior quickly (in less than 20 iterations). *Appbt* and *moldyn* take slightly longer (around 30 iterations). *Dsmc*, however, takes a large number of iterations (around 300) to achieve steady state prediction rates. This is because specific transitions in *dsmc* take a large number of iterations to achieve reasonable prediction accuracies (Table 8).

Memory Requirement of Cosmos Predictors. Table 7 shows that dynamic memory overhead incurred by Cosmos predictors is acceptable—that is, less than 22%—for most applications for predictors with MHR depths of three or lower. Additionally, the number of PHT entries per cache block (or MHR entry) is less than three in most cases. The low PHT to MHR ratio suggests that perhaps a scheme that statically allocates three or four PHT entries per cache block and dynamically allocates the rest from a common pool of memory may work. Only for *barnes* the memory overhead is as high as 63% for MHR depth of three because *barnes* reassigns shared-memory addresses to logically different objects, which confuses Cosmos and leads to greater number of coherence message patterns.

7 Comparison with Directed Optimizations

In this section we compare Cosmos with directed optimizations—that is, optimizations introduced in a coherence protocol for specific sharing patterns. Dynamic self-invalidation [20] and migratory protocols [12, 35] are examples of two such protocols. Both can be thought of as implementing predictors directed at specific optimizations. Cosmos could be less cost-effective than predictors for directed optimizations because Cosmos requires more hardware resources to store, access, and update the Message and Pattern History Tables. However, Cosmos' memory requirement can perhaps be reduced by grouping predictions for multiple cache blocks together (similar to Johnson and Hwu's *macroblocks* [16]).

Cosmos could be better than directed optimizations for two reasons. First, including the composition of predictors of several directed optimizations in a single protocol could be more complex than Cosmos. Predictors in existing coherence protocols are usually integrated with the finite-state machine of the protocol. Such integration may work well when one considers these protocols individually. Unfortunately, combining multiple such predictors into a single protocol can lead to an explosive number of interactions and states, which can make the resulting protocol bulky and hard to debug [11]. More critically,

Transition	**4 iterations**		**80 iterations**		**320 iterations**	
	hits	**refs**	**hits**	**refs**	**hits**	**refs**
`<get_ro_response, upgrade_response>`	2%	20%	34%	4%	62%	2%
`<get_ro_request, inval_rw_response>`	2%	25%	18%	13%	30%	12%
`<inval_rw_response, upgrade_request>`	1%	19%	18%	4%	35%	1%

TABLE 8. *Dsmc*'s prediction accuracies for specific transitions for different number of iterations. refs is percentage of total references to the transition. hits is the percentage of hits to the transition. These numbers are measured with a filterless Cosmos predictor with MHR depth of one.

extending a bulky protocol with other kinds of speculation becomes even harder. In contrast, Cosmos captures the predictors for directed optimizations in a single predictor. Figure 8 shows the coherence message signatures that trigger the dynamic self-invalidation and migratory protocols. Cosmos can capture these signatures easily. Additionally, protocols accelerated with Cosmos are easier to extend because Cosmos separates the predictor from the protocol itself.

Second, Cosmos can not only identify known sharing patterns, but can also discover application-specific patterns not known *a priori*. For example, Section 6.1 shows that one of *unstructured*'s signatures is a complex composition of migratory and producer-consumer sharing patterns. Predictors directed only at migratory or producer-consumer pattern will fail to track *unstructured*'s transition between migratory and producer-consumer sharing patterns. As Section 6.1 also shows, Cosmos can easily capture, filter, and adapt to different message signatures generated by variations in simple sharing patterns studied by Bennett, et al. [7] and Gupta and Weber [13].

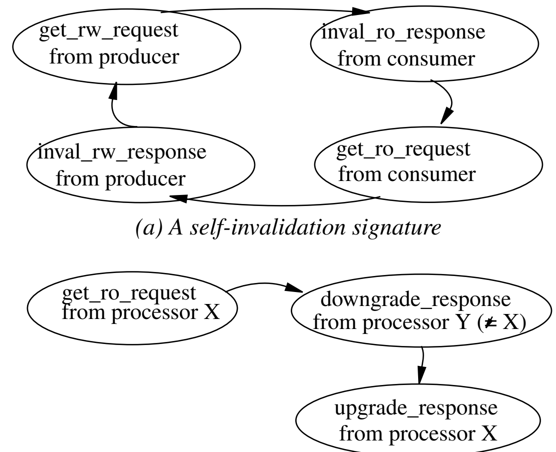
8 Summary and Conclusions

This paper explores using prediction to accelerate coherence protocols. A coherence protocol can execute faster if it can predict future coherence protocol actions and execute them speculatively. It shares with branch prediction the need to have a sophisticated predictor. The first contribution of this paper is the design of the *Cosmos* coherence message predictor. Cosmos predicts the next `<processor,message-type>` in two steps reminiscent of Yeh and Patt's two-level *PAp* branch predictor. Cosmos faces a greater challenge than branch predictors because the Cosmos' prediction is a multi-bit `<processor,message-type>` tuple rather than a single branch outcome bit.

The second contribution of this paper is a detailed evaluation of the Cosmos coherence message predictor. Using five scientific benchmarks on a target shared-memory machine with 16 processors running the Stache directory protocol, variations of Cosmos predict the source and type of the next coherence message with surprisingly-high accuracies of 62-69% (*barnes*), 84-86% (*moldyn*), 84-85% (*appbt*), 74-92% (*unstructured*), and 84-93% (*dsmc*).

Cosmos' high prediction accuracy results from predictable coherence message patterns or *signatures* associated with specific cache block addresses. Such signatures are generated by sharing patterns that do not change or change very slowly during the execution of these applications. Cosmos is more general than directed optimizations, such as dynamic self-invalidation and migratory protocols. Cosmos could be less cost-effective than the directed optimizations because it uses more resources (e.g., tables). Cosmos could be better than directed optimizations because (1) including the composition of these optimizations could be more complex than Cosmos and (2) Cosmos can discover and track application-specific patterns not known *a priori*.

More work is needed to determine whether the high prediction rates



(a) A self-invalidation signature

(b) A migratory protocol signature

FIGURE 8. This figure shows the coherence message signatures that can trigger dynamic self-invalidation (a) and a migratory protocol (b). The downgrade_response, not shown in Table 1, is a response to a downgrade_request sent by the directory. On receiving a downgrade_request for a block, a cache must change the block from exclusive to shared state.

of Cosmos can significantly reduce execution time with a coherence protocol. This is work is analogous to taking a branch predictor with high prediction rates and integrating it into a micro-architecture to see how much it affects the bottom line. We believe that results in this paper on Cosmos' high prediction rates indicate that work on the next step is justified.

Acknowledgments

We would like to thank members of the Wisconsin Wind Tunnel group (`http://www.cs.wisc.edu/~wwt`) for providing an environment conducive to this work and David Wood for providing key feedback on this manuscript. We would also like to thank Rebecca Hoffman, Stefanos Kaxiras, Jim Larus, Avinash Sodani, Dan Sorin, and T. N. Vijaykumar for their helpful comments on earlier drafts of this paper.

References

[1] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 204–215, 1997.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[4] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 261–271, 1997.

[5] Jean-Loup Baer and Tien-Fu Chen. An Effective Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing '91*, pages 176–186, 1991.

[6] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[7] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, June 1990.

[8] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation.

Journal of Computational Chemistry, 4(187), 1983.

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

[10] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[11] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[12] Alan L. Cox and Robert J. 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.

[13] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[14] 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. Earlier version appeared in it Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).

[15] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 179–188, 1995.

[16] Teresa L. Johnson and Wen mei Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, 1997.

[17] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3:79–119, 1988.

[18] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[19] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, 1997.

[20] 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.

[21] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of the Stanford DASH Multiprocessor. Technical Report CSL-TR-89-403, Computer System Laboratory, Stanford University, December 1989.

[22] Tom Lovett and Rusell Clap. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.

[23] Todd. C Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[24] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[25] Shubhendu S. Mukherjee and Mark D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 64–74, Manchester, England, July 1994.

[26] 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.

[27] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[28] Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Detection. Hakan Grahn and Per Stenstrom. *Journal of Parallel and Distributed Computing*, 39(2):158–180, December 1996.

[29] Alain Raynaud, Zheng Zhang, and Josep Torrellas. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, 1996.

[30] 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.

[31] Jonas Skeppstedt and Per Stenstrom. Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 286–296, 1994.

[32] Jonas Skeppstedt and Per Stenstrom. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 69–78, 1995.

[33] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[34] IEEE Computer Society. *IEEE Standard for Scalable Coherent Interface (SCI)*, 1992.

[35] Per Stenstrom, Mats Brorsson, and Lars Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[36] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 191–202, 1997.

[37] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[38] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in it CMG Transactions,/ Spring 1994.

[39] T-Y Yeh and Yale Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.