Multi Markov Predictor: A scheme for Speculative Coherent DSMs

by

T Vinod Kumar Gupta, Angelo Dominguez and Donald Yeung

Department of Electrical and Computer Engineering

University of Maryland

College Park, MD 20742

{tvinod,angelod,yeung}@glue.umd.edu

1 Introduction

Large-scale distributed shared-memory multiprocessors (DSM) have become increasingly popular among designers for high performance parallel machines. DSMs offer a programming environment that is easy to use by providing a logical shared address space over physically distributed memory. They also enhance scalability by removing the shared bus bottleneck in symmetric multiprocessors (SMP). But as the sizes of these machines grow, remote memory latencies increase by several tens or hundreds of processor cycles. These large latencies result from both the increased physical dimensions of the parallel computer, as well as from the increasing clock rates at which the individual processors operate. These large latencies can quickly offset any performance gains from the use of parallelism in executed programs. To address the issue of hiding memory access latencies, caching of shared data is one of the more obvious solutions. While the use of a memory cache in a uniprocessor system has been a popular method for reducing memory latencies in those systems, the use of caches in multiprocessor systems is complicated by the need for coherence among cached shared data. To solve this coherence problem, methods based on directory protocols have been proposed and studied, which maintain a directory entry per memory block that records which processors currently cache the block [9]. Directory based protocols solve the cache

coherency problem, but in most cases these methods can also incur significant latencies in many parallel applications.

To overcome the limitations due to long latencies in directory based protocols, several proposals have been studied that attempt to hide the latencies with current processor work. These techniques can be applied with the help of programmers [10, 11, 12, 13], compilers [14, 15, 16], software [17] or hardware techniques. Hardware techniques include read-modify-write operation prediction in the SGI Origin Protocol [18], pair-wise sharing prediction in SCI [19], dynamic self-invalidation [20] and migratory protocols [21, 22]. Some of the techniques mentioned use only static information but most of these use dynamic information. Furthermore, a protocol implementation is often made more complex by intertwining these techniques with the standard coherence protocol.

The Cosmos predictor [5] proposed by Mukherjee et. al. introduced the field of pattern-based predictors for the first time. Cosmos is based on Yeh and Patt's two-level *PAp* branch predictor [4]. By accurately predicting and performing the necessary coherence operations speculatively in advance, these predictor based DSM can potentially eliminate all of the coherence overhead. The prediction is based on the key observation that much as branches tend to have a repetitive nature leading to accurate predictability, memory blocks also often have a small number of stable, repetitive, and predictable sharing patterns [6]. Moreover, the predictor does not require any change in the standard coherence protocol. Cosmos was followed by *Memory Sharing Predictor (MSP)*, proposed by Lai et. al., that eliminated redundant coherence messages, thereby eliminating mispredictions due to potential re-orderings of acks. Their other contribution was to propose *Virtual MSP (VMSP)*, that eliminates the mispredictions in basic MSP due to re-orderings in read messages.

This paper presents a novel pattern-based predictor, called **Multi Markov Predictor (MMP)**, that improves prediction accuracy as well as coverage and implementation cost over previous approaches. MMP is based on the markov prefetch mechanism [23] for uniprocessors. MMP takes advantage of the key observation that there is correlation between consecutive misses in the cache miss stream of a processor. The MMP predictor applies the markov prediction in the context of DSMs. MMP uses a lookup tables to make future predictions. This direct-mapped table is indexed using the address, access type and

processor id of the current request from a processor. The table provides predictions on next address and access type from the same processor. So while Cosmos and MSP make predictions on the same memory block to be accessed by different processing nodes, MMP makes predictions on the different memory blocks that would be accessed by a processing node after the current access.

Although MSP and Cosmos do a good job at predicting messages, their coverage and accuracy decreases in the presence of message re-orderings and changing sharing patterns. On the contrary, the miss address stream from a processor is often more stable, with little or no re-orderings. This gives an advantage to MMP over MSP and Cosmos. Moreover, the simple structure of the MMP tables offer it lesser inertia than MSP tables to enable them to adapt to changing patterns faster than the earlier predictors. Also, the structure of the tables in MSP and Cosmos scale with increase in number of processors. MMP tables, on the other hand, do not blow up with number of processors.

A para on the results obtained on MMP.

A para on the different subsequent sections.

2 Background

This section describes the structure of a basic directory protocol and reviews previous work on predicting directories in DSMs.

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. This state is typically maintained at a cache block 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 [1].

A cache coherence directory protocol can 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.

Unfortunately, the finite-state machines that implement the coherence logic often incur multiple long-latency operations. 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. This provides the motivation for some kind of prediction methodology that would predict messages in advance so that the penalty due to long latencies of transaction could be reduced.

2.2 Cosmos: A Pattern-Based Message Predictor

Cosmos [5] is a pattern-based coherence message predictor that is derived from the widely-used two-level adaptive PAp branch predictor [4]. Figure 1 depicts the anatomy of Cosmos capturing message sequences for memory blocks at the directory. 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. The number of these tuples is the depth of the MHR.

The second-level table of Cosmos consists of a sequence of *Pattern History Tables (PHTs)*, 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 predictor in figure 1 has a message history depth of one. The figure illustrates an example of possible message sequences for a simple producer/consumer sharing among three processors. Processor(P3) writes to a memory block at address 0x100 and processor 1(P1) and processor 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 in P2 and P1. The pattern table predicts the next incoming message given the

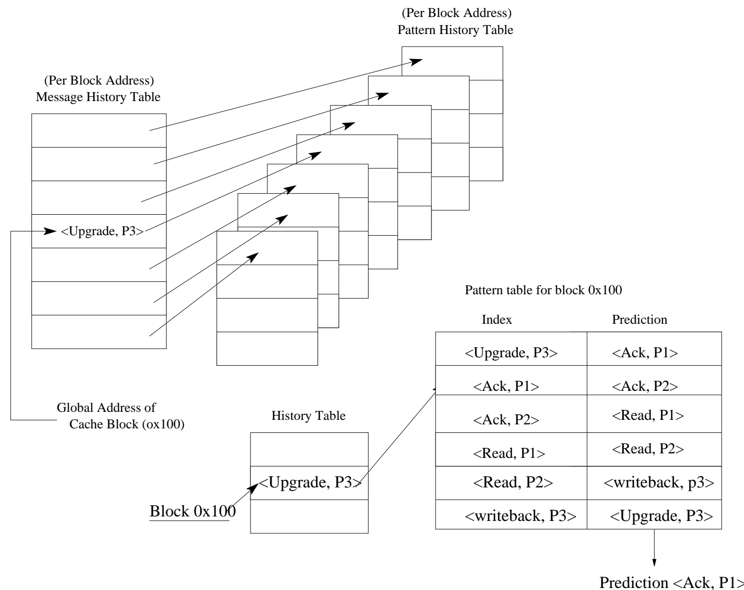

Figure 1: Cosmos two-level message predictor

specific sequence to an acknowledgement by P1. The acknowledgement 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. As in branch predictors, a deeper history enables the predictor to be more accurate by distinguishing among message sequences with common patterns. Although a larger history improves prediction accuracy, it may prohibitively increase the predictor's cost [5]. In the limit, the number of pattern table entries would increase with message history depth. In practice, memory blocks exhibit a small number of stable and distinct sharing patterns [6]. 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.

Although a pattern-based predictor such as Cosmos might do well in the absence of message re-orderings, their coverage and accuracy gets affected substantially in the presence of message re-orderings and changing sharing patterns. The pattern history tables have lot

of inertia in them, that makes them ineffective in the presence of message re-orderings. In order to adapt, all the entries in the table need to be undone and new order needs to be put in to the table. Until then, then predictor will keep mispredicting. For instance, assume that processor P3 is the producer and processors P2 and P1 are the consumers. The table shown in figure 1 captures the scenario where acks and reads from P1 and P2 come in the order of P1 followed by P2. It is possible that this order gets reversed after some time. In the new order, messages from P2 comes before P1. In this case, the first, third, second, fifth and fourth entries in the PHT will get modified to adapt to the new order. The new PHT will look like as shown in figure 2. Thus, a trivial re-ordering has led change in 5 entries. To aggravate this problem, more number of entries in the table imply greater amount of state in the predictor. This would make the tables take much more time to adapt as more information needs to be undone. The table needs to see lot of new patterns to make its state consistent. Moreover, if the re-orderings are notorious enough to toggle between two orders, then it would make the predictor make very few correct predictions.

<Upgrade, P3>	<Ack, P2>
<Ack, P1>	<Read, P2>
<Ack, P2>	<Ack, P1>
<Read, P1>	<Writeback, P3>
<Read, P2>	<Read, P1>
<Writeback, P3>	<Upgrade, P3>

Figure 2: Changed PHT

A DSM may directly implement the history table within the directory because of the fixed amount of required storage for a history entry.

2.3 Memory Sharing Predictors

Memory Sharing Predictors(MSPs) are another class of predictors that work like pattern based ones [8]. An MSP is based on the key observation that to eliminate the coherence overhead on a remote access, it is only necessary to predict the memory request messages(i.e.,

a read, write, or an upgrade). Therefore, MSP gets rid of acks and write-backs. A general message predictor unnecessarily predicts the coherence acknowledgement messages as well, even though these messages are in direct response to a coherence action and are always expected to arrive. Figure 3 illustrates the anatomy of an MSP. The MSP eliminates all the sequences resulting from the potential re-orderings of the acknowledgements. The MSP requires two bits to encode three request message types(read, write and upgrade) as compared to a message predictor requiring three bits to encode three request types and two acknowledgement types(i.e., response to read-only invalidations and write-backs).

Apart from getting rid of redundant coherence messages, MSP also proposes *Vector MSP (VMSP)*, that keeps track of all the readers of memory block rather than the order in which they read. This enables the predictor to get rid of re-orderings in the read sequences.

Elimination of acks and read request re-ordering in MSP decreases it's inertia but not enough for it to easily adapt to changing sharing patterns. For instance, assume that processor P1 is the producer and processors P2 and P3 are the consumers. After some time, processor P4 becomes the producer and processors P5 and P6 become the consumers. This would lead to cold start misses. Figure 4 refers to the initial PHT and figure **??** refers to the changed PHT. Processor P5 becomes the producer after P3 has read the block. The entries change in the order: third, fourth, first and second. The example demonstrated one misprediction and three cold start misses. These cold start misses and mispredictions decrease the coverage and accuracy of the predictor. It is noteworthy to observe here is that these inertia misses are independent of the advantages that MSP has over Cosmos. The example demonstrates the inertia in the state of the predictor that render it slow in responding to changing sharing patterns.

2.4 Markov Predictor based Prefetching

Markov prefetching (refer to markov paper) is one of the prefetching techniques that are used in uniprocessors to reduce the memory stalls that occur due to cache misses incurred. Different prefetching mechanisms attempt to predict the next memory access to be made by the processor and then fetch it from the main memory before the processor makes request for that cache block. This hides the data transfer time, thereby reducing the memory overhead

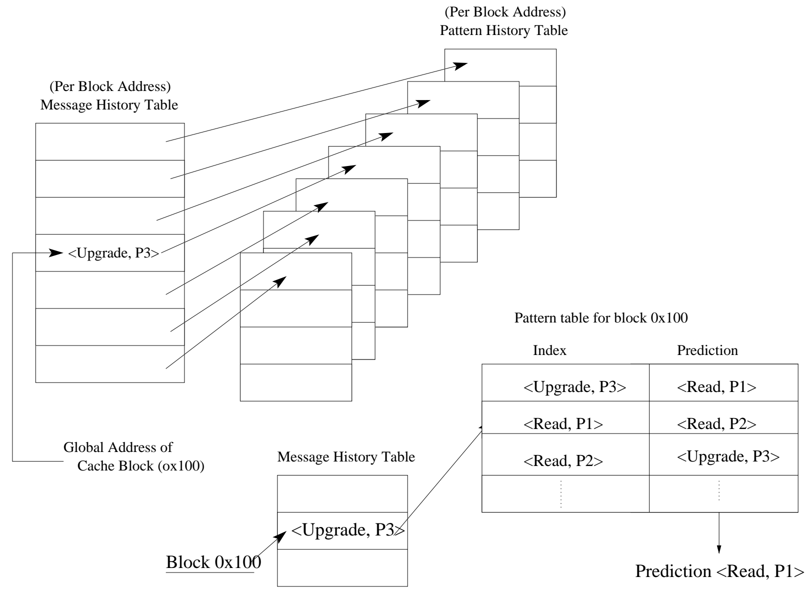

Figure 3: A Memory Sharing Predictor

<Write, P1>	<Read, P2>
<Read, P1>	<Read, P3>
<Read, P3>	<Write, P1>

Figure 4: Initial PHT in MSP

<Read, P5>	<Read, P6>
<Read, P6>	<Write, P4>
<Read, P3>	<Write, P4>
<Write, P4>	<Read, P5>

Figure 5: Changed PHT in MSP

in the cycles per instruction. Markov predictor is one of the prefetching mechanisms used to hide memory latencies. It is based on the observation that the address stream seen in a program can be efficiently modeled by a Markov model. A Markov model is a set of states and transition frequencies where each state has a probability of transition to another. The markov prefetcher models miss addresses as states and transition frequency as the probability that the destination node misses after the source node misses.

Based on the states and state transition frequencies, a table is built as shown in figure 6. In this table, each state in the markov model occupies a single line in the prediction table, and can have up to two to four transitions to other states. These transition addresses are the address predictions that the prefetcher makes when it sees a miss. The predictions are ordered in decreasing order of their probabilities. Hence, the probability of the first prediction being correct is higher than the second, the second being higher than the third and so on. When the current miss address matches the index address in the prefetch table, all of the next predicted addresses are eligible to be prefetched.

Current Miss Reference Address	Next Address Prediction Registers			
Miss Addr 1	Pred 1	Pred 2	Pred 3	Pred 4
–––––	–	–	–	–
–––––	–	–	–	–
Miss Addr N	Pred 1	Pred 2	Pred 3	Pred 4

Figure 6: Table representing Markov prediction.

3 Multi Markov Predictor (MMP)

This paper proposes a new class of pattern-based predictors called the *Multi Markov Predictors* (MMPs). An MMP is based on the key observation that there is a correlation among the memory requests made by a given processor, as seen by the directory in a DSM. In other words, programs access main memory using a particular pattern or access model. Once such an access model has been determined, hardware mechanisms like the MMP can be designed to capture that reference stream. MMP makes use of the markov model [23] that is available for uniprocessors and applies it to a DSM system.

3.1 Basic Structure of MMP

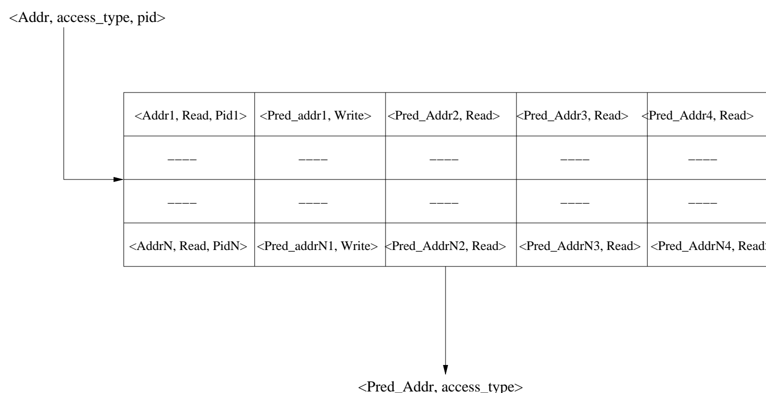

Figure 7: MMP Predictor

The structure of an MMP is illustrated in figure 7. The MMP sits and operates besides each directory in a DSM. The directory in a DSM is responsible for fulfilling cache miss requests from individual processors by executing the directory protocol [1]. As seen in the figure, the MMP maintains a table called the MMP table, that contains a list of entries. Each entry has a triple (3-tuple) in its index column. The triple consists of the memory

address of the requests seen by the directory from a given processor, along with its access type and processor id of the processor that made the request. The access type in this case is either read or write. The MMP uses these triples as a tag to index into the table. The number of such triples used to index into the table is the history depth of the MMP. In the figure, the history depth is one. Against each of the triples, there is a set of tuples, which are the predictions for the next memory block address that would be requested by the same processor, along with its access type. These prediction tuples are ordered in decreasing order of their probabilities.

The MMP table for the predictor is a fixed size direct-mapped table, that is indexed using the lower order bits of the bit pattern obtained from the concatenation of address, access type and processor id of the requesting processor. This triple also occupies the index column of the row to perform tag matching, as different triples could map to the same row. Unlike Cosmos and MSP, this table is not part of the directory structure due to space considerations. Each entry in the table needs several bits to store predictions and frequencies, thereby making it infeasible to integrate it to the directory entry. Moreover, entries corresponding to a large number of memory blocks would be vacant, leading to wastage of space. On the other hand, having an independent table offers a degree of parallelism to make predictions in the memory controller. Details of these analysis is presented in section 4.

An illustration of the structure is as follows. The table shown in figure 7 has a row with ¡addr1, read, pid1¿ in the index column. The next four tuples in the row correspond to the succeeding tuples that the directory had seen in the past, following the current triple. ¡pred_addr1, write¿ tuple has the highest probability followed by others in the row, i.e. the MMP observed this tuple the maximum number of times following the current triple. Thus MMP will predict all or some of the the non-vacant entries in the row, which in this case is $< pred_addr1, write >$, $< pred_addr2, read >$, $< pred_addr3, read >$, $< pred_addr4, read >$.

The access types used by MMP are read and write only, as it considers only these two accesses to index into the tables and makes predictions only about reads and writes. The reason for doing so is that other access types, namely, invalidations and acknowledgements, are the result of read and write operations. Hence, they are always expected to arrive in

response to reads and writes and, consequently, there is no need to store their prediction. Furthermore, coherence messages seen by the directory do not necessarily arrive in the same order owing to non-uniform processor occupancies and changing network loading. Re-orderings of these messages will call for a larger depth in the table and also larger number of predictions to be made to obtain the same prediction performance. Also, tracking all the coherence messages will require more bits to encode the message types recorded by the predictor. In order to get rid of all these inefficiencies, MMP predicts only those messages that designate read and write messages by processors for different memory blocks, thereby reducing the number of bits needed to store the predictor table.

The process id of the requesting processor is used to index in to the MMP table. This is done in order to model the predictor as set of independent predictors, that are predicting on future messages coming from processors corresponding to them. MMP predictor takes advantage of the repetition in the message streams from a single processor. Therefore, the processor id field is necessary. This also explains for the fact that there is no need to have the processor id component in the prediction tuples. The prediction tuples are the predictions for the processor whose id was used to index in to the table. Hence, having processor id in the prediction tuple would be redundant.

Sections 3.2 and 3.3 provide details of the exact steps involved in obtaining a prediction from and updating MMP respectively.

3.2 Obtaining Predictions from MMP

Here are the steps involved to obtain a prediction from MMP.

- Index into the MMP table, using the lower order bits of the bit vector obtained from the concatenation of the current request address, its access type and the processor id of the processor that made the request.

- If the entry is found in the table, then return all the non-vacant prediction tuples. In a table with four predictions, the number of predicted tuples (next address and access type) is at most four. It is to be noted that these predictions correspond to the processor that made the current request.

3.3 Updating MMP

The MMP predictor updates its table after it receives every message. Here are the steps involved in updating the MMP table:

- Index into the MMP table, using the lower order bits of the bit vector obtained from the concatenation of the previous request address, its access type and the processor id of the processor that made the current request.
- If an entry is found in the table, check to see if the current tuple, i.e. the tuple of current request address and its access type, is one of the predicted tuples. If yes, update the probability of the current tuple. Sort the predictions if needed, i.e. the updated probability of the current tuple exceeds the probability of the preceding prediction tuple.
- if an entry is found in the table but the current tuple, of current request address and access type, is not in the predicted tuples, then check to see if there is an empty slot. If an empty slot is found, fill in the current tuple, after updating its probability. But if there is no empty slot, then the current tuple replaces the last prediction tuple.
- if the desired entry is not found, then create a new entry with the previous request triple as the index, and the current request tuple as the first prediction tuple.

3.4 How MMP Differs from Markov Prefetchers

MMP differs from markov based prefetching mechanisms [23] in two ways. First, the MMP table is indexed by a triple of address, access type and processor id as against a markov predictor where the indexing is done by just the address. This is necessary because a prediction made based solely on the address would not be able to differentiate between different reference streams corresponding to different access types, assuming that they come from the same processor. Hence, it would limit the accuracy of the predictor. The requests that a processor makes after reading a shared block might not be the same as those it makes after it writes. This phenomenon is expected to be more pronounced for multiprocessors because most shared-memory parallel applications tend to read in shared data, do local

computations and then subsequently write them. The basis of such a execution model is that it leads to decrease in the contention for locks.

The second point of difference between MMP and markov prefetchers is that the prediction resources in MMP are present next to the directory, i.e. the main memory, as opposed to markov prefetcher, whose resources sit next to the L1 cache (i.e. in between L1 and L2 cache). Having the predictor coordinate with the directory leads to three advantages. Firstly, the memory bandwidth utilized due to such a setup is halved as compared to having it in the processor node. In this present case, the predictions are made in the directory and appropriate protocol actions are carried out directly. On the other hand, if the predictor were in the processor, then all these predicted requests need to be sent to the directory to be carried out. Hence, instead of a two-way trip, there is only a one-way trip now. The second advantage is that the implementation cost of the predictor is lower, if done in the main memory as opposed to processor node. This is primarily due to the fact that processor area needs to be minimal from power, cost and fabrication points of view. Hence, it is logical to keep the predictor with in the memory modules. The third advantage is that it would be easier to implement MMP in the memory modules rather than the processor. The reason for this is that the predictions made would require the directory to initiate coherence actions on its part like sending invalidations to other processors. This is easier to do if the directory has the prediction. If the processor is to make this prediction, then somehow, it has to be conveyed to the directory to invalidate a cache block in another processor and that this is a prediction and not a request. This might require modifying the directory protocol which is not desirable.

3.5 How and Why MMP Works?

MMP predictor captures the repetition in the cache miss stream of a processor to give good prediction accuracies as well as good coverage, when compared to other pattern based predictors [5, 8]. In most of the applications, especially parallelized scientific applications, there exits a pattern in which a given processor requests for cache blocks. This pattern is effectively captured by our MMP predictor.

To illustrate the repetition in the memory accesses made by a processor node in a

DSM application, a benchmark application is analyzed. The application is the "Water" application from the SPLASH suite [24]. Water is a N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a user-specified number of time-steps. Every time-step involves setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions, using Gear's sixth-order predictor-corrector method [25]. The total potential is computed as the sum of intra- and intermolecular potentials.

In order to execute the above solvers, the application makes use of some data structures. The main data structure for this application is a large, one-dimensional array VAR. The array is divided into eight sections, each representing a variable in the system, namely the forces and their derivatives and for resultant force. The execution of the application is divided into a set of sequential tasks, which are divided among the processors. Each processor is assigned a set of molecules that are located next to each other in the VAR array. Thus a given processor is responsible for all the force and displacement computations for its own molecules.

The MMP predictor presented in this paper exploits the nature of the data access pattern in the above application. The data access pattern is completely predictable, both within and across time-steps. Since each processor accesses the same portion of the array VAR to perform similar computations across different iterations, the MMP predictor table reaches a stable state after few iterations. Thereafter, the memory accesses made by the different processors in subsequent iterations are correctly predicted.

Although the data access pattern in Water is predictable, the sharing of molecules is tremendous. Each processor reads in values from the next $(p/2)$ processors. This implies that each molecule is accessed by $(p/2)$ processors. Clearly, the sharing scales with the increase in machine size and the limited depth of the PHTs would render MSP and Cosmos unable to capture the sharing. This would lead to bad prediction accuracies. Even in the absence of any limit on PHT depth, a minor re-ordering of the read messages would decrease the performance. Although, this problem could be addressed by the VMSP mechanism, the space requirements and complexities of VMSP as well as the scalability factor of the problem

would offset its advantages.

The MMP predictor can also adapt to complex message streams that are unlike the above "Water" application, which is an example of static problem partitioning and static load balancing. Complex message streams display dynamic load balancing and changing problem partitions. An example of such an application is "Barnes-Hut", which simulates the forces among the bodies in a galaxy. This application displays sharing patterns which change after every iteration due to restructuring of the data structures involved. Please refer to section (refer to barnes part of results) for further details. The MMP predictor works for even such kind of applications because it can quickly sense the change in access patterns and update its table. The table structure in MMP is quite simple and flexible to adapt to such situations. Whenever there is a change in the access pattern, the update method of MMP ensures that the new address pattern is captured in the table. If the new pattern becomes predominant, then its frequency in the table gets incremented. Eventually, it would move up in the prediction list. On the contrary, if the new pattern is a temporary change, then it would get evicted from the prediction tuples, whenever the new pattern comes. This flexibility is absent in other predictors, MSP [8] and Cosmos [5], which need to undo a considerable amount of their prediction data to adapt to the new configuration.

4 MMP Versus Other Predictors

The MMP predictor differs from the earlier pattern-based predictors, namely, MSP and Cosmos in different ways. It differs in the basis on which repetition in the message stream to a memory controller can be captured to predict, i.e. the philosophy behind the prediction mechanism. From an anatomical standpoint, the two differ. Consequently, memory requirements of the two differ. The two also differ in terms of their adaptability to complex situations. The following sections discuss each of these in detail.

4.1 Mechanism Differences

The MMP predictor differs from the other pattern-based predictors, namely, MSP and Cosmos, in the manner in which patterns are extracted. MSP and Cosmos take advantage of

the repetition in the sharing pattern that occurs in a parallel program for a given memory block. Different processors take turns to access a memory block in a repetitive and predictable manner. This explains the basis for the working of MSP and Cosmos. Both these predictors have a mechanism to track and capture such repetitions in their data structures. MSP's data structures do the prediction for the next processor node that would request the current request memory block, along with the access type.

On the other hand, MMP takes advantage of the fact that the memory accesses made by a processor to the shared memory modules in a DSM is repetitive in nature. Program codes, specially scientific codes that call for the processing capabilities of multiprocessors, demonstrate lot of repetition in their memory access patterns. Most applications execute a block of computation for certain number of iterations. Within each block of execution, they tend to make similar memory references in a predictable order. It is this repetition that MMP attempts to exploit. Moreover, the repetition is expected to be more prominent and more capturable in DSMs than uniprocessors. The reason is that the sharing of data between processors in a DSM leads to frequent invalidations of the memory blocks in the processors. Hence, the probability of a repeat in the miss pattern is high.

4.2 Anatomical Differences

MMP predictor differs from MSP and Cosmos structurally. MSP and Cosmos integrate their prediction tables and tracking data structures with the directory structure. The directory entry in a directory protocol maintains its sharing information along with the memory block. MSP and Cosmos, add to this information, by storing their message history registers and pattern history tables, along with the directory entry. However, MSP and Cosmos could also be implemented independent of the directory entry. But the thread of commonality is that they need MHRs and PHTs corresponding to each memory block. Hence, for each memory block there is a data structure to do prediction.

MMP predictor, however, maintains a separate table to store and make predictions. The table is indexed using the concatenation bit-vector of address, access type and processor id. The table contains prediction tuples for subsequent accesses by the same processor. This table contains lesser number of entries than the number of memory blocks. The need for

such a table structure arises due to space considerations. Section (refer to memory section) discusses the memory implications of MMP.

As regards to the hierarchical structure of the tables, MMP uses a flat one level table structure. On the other hand, MSP and Cosmos use a hierarchical two level table structure. The first level is to index into the correct memory block and the next level is to index into the correct entry in the PHT.

4.3 Memory Difference

The amount of memory required for the MMP predictor is derived as follows:

$$Mem_for_MMP = (Num_of_Table_Entries) * ((32 + \log(Num_of_Access_Types) + \log(NPROCS)) + (Num_of_Predictions * (32 + \log(Num_of_Access_Types) + \log(Maximum_Frequency))))$$

In the above equation, NPROCS refers to the number of processors in the system. The Num_of_Table_Entries term captures the number of entries in the table. Each entry, has a triple of the size of $(32 + \log(Num_of_Access_Types) + \log(NPROCS)$. The entry also has Num_of_Predictions number of prediction tuples. The size of each such tuple is $(32 + \log(Num_of_Access_Types) + \log(Maximum_Frequency))$. Each tuple consists of an address, which is assumed to be 32 bits and access type. Apart from this, each prediction tuple also keeps track of the frequency of the predictions. The total size of the table is the size of each entry multiplied by the number of such entries.

The following equation represents the memory required for the MSP predictor.

$$Mem_for_MSP = (Num_of_Cache_Blocks) * ((\log(NPROCS) + \log(Num_of_Access_Types)) + ((PHT_Depth) * 2 * (\log(NPROCS) + \log(Num_of_Access_Types))))$$

$$Num_of_Cache_Blocks = \frac{Mem_Size_at_Each_Module}{Cache_Block_Size}$$

Each memory block in a MSP predictor has a message history register (MHR). The size of this register is $(\log(NPROCS) + \log(Num_of_Access_Types))$. The MHR points to a pattern history table (PHT). The number of entries in the PHT is equal to its depth. Each entry in the PHT has a pair of tuples of processor id and access type. Adding the individual sizes of these elements derive the equations above.

In order to compare the memory requirements of the MSP and MMP schemes, the memory requirement equations of both are equated. This would determine the number of

entries in the MMP tables, such that the memory required for the two is same. Assuming that the number of processors is 16, cache block size is 32 bytes, depth of PHT is 4, memory size of each module is 8MB, number of access types is 3 and 2 for MSP and MMP respectively, number of predictions made by MMP is 2 and number of bits required for maximum frequency is 20, the number of entries derived is 10^5 approximately. The results (refer to section results), shows that the average number of entries required for MMP predictor is roughly half of what is required by MSP. This analysis shows that the MMP predictor is more efficient than MSP in terms of memory requirements.

4.4 Theoretical Differences

From a theoretical point of view, MMP and MSP predictors are orthogonal to each other. Upon a memory request from a processor, the MMP predictor predicts the address and the access type of the next memory request expected from the same processor. MSP, on the other hand, predicts the next processor and the access type that would request the given address. In a DSM memory module, there are 3 variables involved to make any kind of prediction: the processor id, memory block address and access type. MSP predictor fixes memory block address and makes prediction on the next processor id and access type. MMP predictor fixes processor id and makes prediction on the next memory address and access type. MSP technique would be advantageous in the case of locks. Several processors attempt to acquire the lock to enter critical regions. Locks become a performance bottleneck since they stall processors for long periods of time. But MSP technique could be used to speedup the movement of locks among the contending processors. This can be helpful in speeding up applications. MMP would be advantageous in speeding up the computation regions of the applications. Since MMP predictor predicts future memory requests from the same processor, they can be helpful in decreasing the cache misses. Consequently, this technique speeds up the application.

A disadvantage of MSP is that the penalty of mispredictions could be high. By mispredicting a message, the MSP might evict the cache block from several processors, although it might still be needed by one of the processor. This would require undoing all actions. This technique can, hence, back fire, especially during critical regions. Mispredictions by

MMP, however, might not be as fatal as MSP. The reason is that MMP predicts the next memory request from the same processor to the same directory controller. In reality, this might not be the next miss that the processor sees. Subsequent misses could be to other directories. In such a case, there is enough time for the directory to undo a misprediction.

5 Methodology

To evaluate the effectiveness of MMP in predicting coherence messages in a DSM, several shared-memory applications were executed on NWO, a cycle accurate simulator for the MIT Alewife machine[**?**]. NWO was used to simulate a 16-node implementation of the Alewife and gather memory access traces from the benchmark programs studied. The traces gathered were processed by a directory simulator implemented as part of this study in order to produce coherence message miss streams for use in MMP. This simulator is parametrizable, and based on a full-map and write-invalidate directory protocol. Table 8 depicts the directory simulation parameters used.

In order to gather meaningful coherence message miss streams from the directory simulator, perfect instruction caches are assumed, although realistic data caches are implemented. Capacity, compulsory and conflict cache misses are modeled by the directory simulator. Because of the unpredictability of conflict cache evictions, these are an important class of directory requests which are generally difficult to predict. The other major obstacle in predicting messages for a DSM machine is that of memory reference reordering, and this effect is directly recorded in the memory traces obtained from NWO. The directory simulator assumes blocking caches on multiple directory requests, although the directories are assumed to handle requests immediately from the perspective of the processor. It is further assumed that the network is congestion and contention free, and does not interfere with the coherence messages seen by the directory. The simulator produces the coherence requests seen for each processor's shared memory module.

The benchmark applications studied and their corresponding input parameters are listed in Table 9. *MP3D* and *Water* are from the SPLASH benchmark suite[24], while *Barnes* is from the SPLASH-2 suite[**?**]. *Em3d* is a shared-memory implementation of the Split-C

benchmark. MTV is a public-domain raytacer ported to the Alewife platform. Each of these programs illustrates different behaviors that are commonly seen in parallel applications ported to a DSM. Some details on the algorithms and behaviors observed for each application are discussed to illustrate their effect on prediction accuracies for MMP.

Number of Nodes	16
Cache Size per node	1 MByte
Memory per node	4 MBytes
Cache block size	32 bytes
Associativity	1

Figure 8: DSM simulation parameters

Application	Input Parameters	Iterations
Mp3d	20,000 molecules	10
Barnes	4096 particles	21
Em3d	78,600 nodes, 15% remote	50
Water	216 particles	50
MTV	balls.nff, 600x600 resolution	n/a

Figure 9: Applications and input parameters

6 Results

In this section, the prediction accuracy of MMP for each of the benchmark programs under study is presented, followed by further insights on the mechanisms that make those predictions possible in each case. To evaluate the performance of MMP against other pattern-based message predictors, results are shown for an implementation of MSP, the optimized pattern-based message predictor discussed previously. MSP was implemented with an unlimited history depth, to show its best-case performance. For each of the benchmarks run, the parallel portions of the code were used to gather traces, and predictor results reflect coldstart and initialization behavior as well as the steady-state patterns captured. The

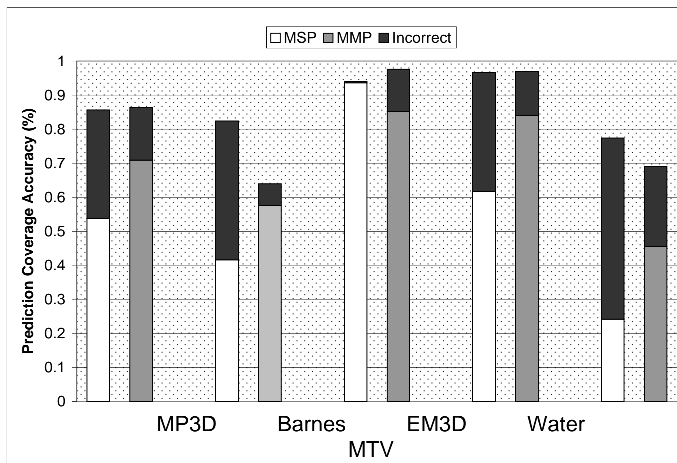

Figure 10: Fraction of directory requests covered by predictions

results presented show that MMP can significantly improve prediction accuracy over MSP for classes of applications which do not employ the message patterns that MSP exploits. It is further shown that MMP offers comparable, if not better performance on other types of parallel benchmarks.

6.1 Prediction Accuracy

Figure 10 shows the performance of MSP and MMP, where MMP was given a prediction depth of four entries. For each application, the percentage of correct and incorrect predictions is shown, relative to the coverage each predictor achieved for the reference stream observed. The figure shows that the coverage accuracy of MMP is higher than 70 in three of the five applications, and outperforms MSP in four of the five benchmarks. The prediction method in MMP allows it to perform well, even for programs with changing data sharing patterns, by adapting quickly and taking advantage of the iterative method that most parallel programs employ.

MP3D simulates the trajectories of a collection of molecules representative of continuous fluid mediums, until finding a steady-state for the model. The molecules are partitioned statically, assigning each processor an equal number of molecules. As the molecules move

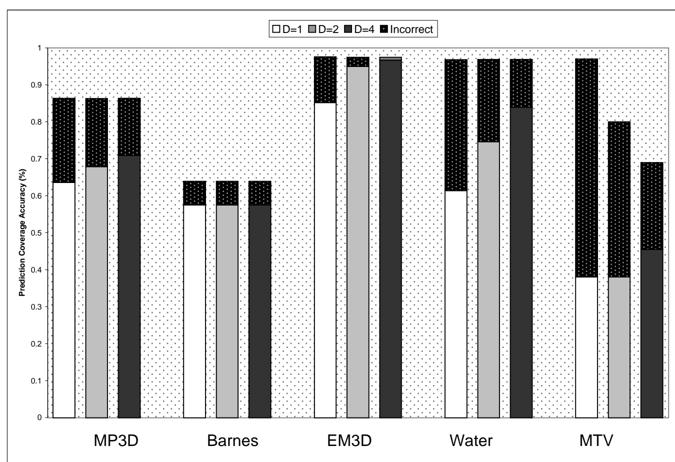

Figure 11: MMP Accuracy with varying prediction depth

in space, their interaction with molecules from other processors varies as a result. Barriers are used for synchronization, while little locking activity usually takes place. The changing sharing patterns, and lack of stable migratory-sharing can cause problems for general message predictors. MMP adapts quickly to the changing interaction patterns, which usually remain stable for a number of iterations, and takes advantage of the repetitive order in which each processor accesses its own molecules, and consequently requests data from other processors at the same points during execution. MMP achieves an accuracy greater than 70 in correct predictions on future processor requests for data.

Barnes is a classical N-body simulation, where each body is modelled as a point mass interacting through gravitational forces with other bodies. In it, a heirarchical octree representing the bodies in space is traversed to calculate the forces for a given body during each step. This octree is rebuilt during each step, resulting in rapidly changing sharing patterns between processors. This process makes it more difficult for MMP to recognize patterns in the addresses accessed between iterations, although its correlation between successive addresses accessed is formed quickly, and enables the predictor to a high ratio between correct and incorrect predictions, of more than X percent.

EM3D in em3d, a node interacts with a random number of nodes. Of them, a percentage are remote, which is a parameter to the program. The sharing pattern is quite stable, due to stable producer/consumer pattern. hence msp results are good.. same holds true for mmp also.. the pattten for a given processor is also stable. Static data structures and equal allocation show strong producer-consumer sharing patterns, with a constant repetitive traversal of the data structures emplyed. Both MSP and MMP take advantage of the stabilirt of this behavior and issue a large number of correct predictions.

Water Some analysis earlier, refer to that. Water is an N-body molecular dynamis application similar to MP3D. Spatial locality is exploited for data assignment when assigning molecules that are next to each other in the main array.

MTV MTV is a classic raytracer, which lent itself to easy parallelization for the DSM environment due to its nature of computation. For a given input scene to be rendered, MTV calculates the paths of light rays sent from a specified coordinate to all points in the scene. Each processor is assigned an approximately equal section of the scene to handle, and interacts with other processors depending on the paths of the rays through the scene. While both MMP and MSP do not excel at making correct predictions, due to the pseudo-random traversal of the scenes depending on these paths, MMP does more easily capture the repetitive nature of pixel sharing between processors, due to the nature of rays as they traverse space. This type of application is often common of programs ported quickly to a parallel environment, and hiding message latency through predictions becomes all the more important to achieve high performance.

6.2 Prediction Depth

6.3 Prediction Timing

References

[1] 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.

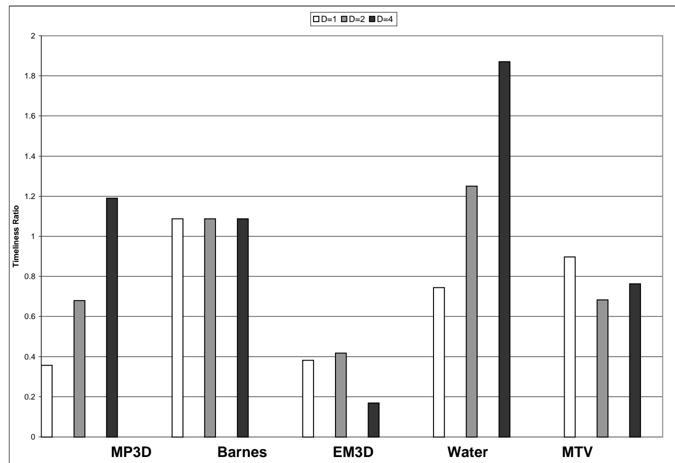

Figure 12: Timeliness of MMP predictions

[2] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In proceedings of the 21st Annual Symposium on Computer Architecture, pages325-337, April 1994.

[3] Tom Lovett and Russell Clap. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In proceedings of the 23rd Annual International Symposium on Computer Architecture, pages308-317, 1996.

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

[5] Shubhendu S. Mujherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocols. In proceedings of the 25th Annual International Symposium on Computer Architecture, June 1998.

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

[7] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and WolfDietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. Proceedings of the 18th International Conference on Computer Architecture, pages 254-263, IEEE, May 1991.

[8] An-Chow Lai and Babak Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In proceedings of the 26th Annual International Symposium on Computer Architecture, 1999.

[9] Chaiken et al., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," Computer, 23(6), pp. 49-58,June 1990.

[10] 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.

[11] 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.

[12] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communications in Cache-Coherent Multiprocessors. In Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, pages 204-215, 1997.

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

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

[15] Jonas Skeppstedt and Per Stenstrom. Simple Compiler Algorithms to Reduce Read Latency in Ownership-based Cache Coherence Protocols. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pages 69-78, 1995.

[16] Tor E. Jeremiassen and Susan J. Eggers. Reduce False Sharing in Shared Memory Multiprocessors. In 5th ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming(PPOPP), pages 179-188, 1995.

[17] 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 3rd IEEE Symposium on High-Performance Computer Architecture, pages 261-271, 1997.

[18] 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.

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

[20] Alvin R. Lebeck and Favid A, Wood. Dynamic Self-Invalidation: Reducing Cohernece Overhead in Shared-Memory Multiprocessors . In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 48-59, June 1995.

[21] 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, 1993.

[22] Per Stenstrom, Matts 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, 1993.

[23] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 252-263, 1997.

[24] Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[25] C.W.Gear. Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, New Jersey, 1971.