Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors[1]

Alain Raynaud, **Zheng Zhang**, and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
raynaud,zzhang,torrella@csrd.uiuc.edu

Abstract

While update protocols generally induce lower miss rates than invalidate protocols, they tend to generate much traffic. This is one of the reasons why they are considered less cost-effectively scalable than invalidate protocols and, as a result, are avoided in most existing designs of scalable shared-memory multiprocessors. However, given the increasing relative cost of cache misses, update protocols are becoming more worthy of exploration.

In this paper, we present a model of sharing that is key to investigating the performance of optimized update protocols: the *Update Distance Model*. The model gives insight into the update patterns that optimized protocols need to handle. Using this model, we design a new family of protocols that we call *Distance-Adaptive Protocols*. In these schemes, the directory records the update patterns observed and then uses them to selectively send updates and invalidations to processors. As a result, traffic and miss rates are kept low. We present an implementation of these protocols based on a dynamic pointer scheme. A performance comparison between one of these protocols and efficient invalidate and delayed competitive-update protocols over five applications shows that the new protocol decreases the execution time by an average of 15% and 10% respectively.

1 Introduction

While hardware-coherent scalable shared-memory multiprocessors can use invalidate or update messages to ensure that processors do not access stale data, most existing designs of these machines use invalidate schemes. This is due to the belief that update protocols are less cost-effectively scalable than invalidate protocols.

One of the major problems of update protocols is that they tend to generate much inter-processor traffic. Such traffic can clog the network of the machine and slow down the execution of applications. Invalidate protocols, instead, do not send update messages and, therefore, usually keep inter-processor traffic low. However, since caches are invalidated, processors miss when they re-reference data. Nevertheless, it is felt that a higher miss rate for invalidate protocols is an acceptable cost to pay in order to avoid high volumes of traffic. Moreover, to lessen the cost of this higher miss rate, several techniques have been proposed to help the processor hide or tolerate the miss penalties. These techniques include prefetching, multithreading, and relaxed memory consistency models among others. The result is that the performance impact of the misses has been reduced significantly.

Unfortunately, as memory access latencies continue to increase relative to processor cycle time, keeping the number of misses low becomes increasingly important. Indeed, even infrequent instances of non-hidden cache misses can result in low processor utilization. Consequently, update protocols are now becoming more worthy of exploration as an alternative. This is especially true if we can design update protocols with low traffic requirements.

In recent years, several techniques have been proposed to reduce the traffic in update protocols. The first one is to add a competitive threshold to the updates [7]: after a certain threshold number of updates have been received by a cache whose processor has not used the data in the meantime, the data is invalidated. Obviously, the threshold used directly influences the resulting traffic. A study by Nilsson *et al* [10] found that a threshold of four is an optimal choice. A second study by Dahlgren and Stenstrom [2] using write caches in the context of relaxed consistency concluded that a threshold of one is sufficient.

A second technique to reduce traffic is update combining. Basic combining of updates has been proposed by Glasco *et al* [5]. One class of protocols that provides opportunities for combining is delayed protocols, proposed and described by Dubois *et al* [3] in the context of invalidate protocols. Write caches constitute one implementation of combining [2]. A final technique to reduce traffic is to extend migratory data detection and handling to update protocols. This has been presented by Nilsson and Stenstrom [9]. The idea is to use an invalidate protocol for migratory data and an update protocol for the rest. This scheme reduces the traffic because migratory data is particularly traffic-intensive under update protocols.

Some of these techniques have been proposed separately; they have not all been combined in a single protocol and then compared to invalidate protocols. Much more importantly, however, there is no framework that illuminates the features that advanced update protocols need to provide to successfully minimize misses and traffic. Indeed, previous models like Eggers and Katz's write run model [4] do not necessarily provide such insights for advanced protocols like delayed update protocols.

In this paper, we present a new model of sharing that gives such insights: the *Update Distance* model. Then, using the model, we design a new family of scalable update protocols called *Distance-Adaptive Protocols*. In these schemes, the directory records the update patterns observed and then uses them to selectively send updates and invalidations to processors. We present an implementation of these protocols based on a dynamic pointer directory scheme. Finally, we compare the new protocols to existing protocols. The new schemes decrease the execution time of five applications by an average of 15% and 10% over efficient invalidate and delayed competitive-update protocols respectively.

This paper is organized in four parts: Section 2 discusses the experimental system used for this research; Section 3 presents the Update Distance model and applies it to our applications; Section 4 uses insights from the model to introduce the Distance-Adaptive protocols and presents an im-

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098; ARPA Contract No. DABT63-95-C-0097; and NASA Contract No. NAG-1-613.

Table 1: Application characteristics.

Application	Description	Input Data Set	Type of Data Sharing
MP3D	Simulates a rarefied hypersonic flow	10,000 particles for 10 steps	Mig
Water	Simulates evolution of set of water molecules	343 molecules for 6 steps	Mig
Ocean	Simulates eddy currents in an ocean basin	50 by 50 grid	P-C
Barnes	Simulates evolution of galaxies	1,000 bodies for 6 steps	Mix
Pthor	Simulates a digital circuit at the logic level	Small RISC processor	Mix

plementation of them; finally, Section 5 evaluates the performance of the protocols.

2 Experimental System

The quantitative analyses presented in this paper use execution-driven simulations of parallel applications. Since all sections of the paper refer to the applications and simulation system used, we introduce them first in this section.

2.1 Applications

We use a subset of the parallel applications in the Splash benchmark suite [12]. Table 1 describes what they do and their input data sizes. The fourth column shows the most frequent type of data sharing pattern in the applications. The patterns can be migratory (Mig), producer-consumer (P-C), or a mixture of several patterns (Mix).

2.2 Simulation Environment

We perform detailed execution-driven simulations of a shared-memory multiprocessor. We use TangoLite [6] to simulate a 24-node NUMA multiprocessor. Each node in the machine contains a 100 MHz processor, a 4-Kbyte primary cache, an infinite secondary cache, a write buffer with 16 entries after the secondary cache, a write cache with 4 entries in parallel with the secondary cache, and a portion of the global memory with its corresponding directory. The primary cache is direct-mapped and write-through. The secondary cache is write back and lock-up free. We use an infinite secondary cache to de-emphasize conflict misses. The cache coherence protocol used is the Illinois protocol. Write misses do not stall the processor, while read misses stall the processor and bypass the pending writes. We use the release consistency model. The network is a 32-bit wide bidirectional mesh.

While only shared data is simulated, all contention in the machine is modeled, including the network. For the network, we use the model proposed in [1]. The machine has the following read latencies without contention: 1 cycle for a primary cache hit, 15 cycles for a secondary cache hit, an average of 120 cycles for a 2-hop global memory access and an average of 180 cycles for a 3-hop global memory access. The coherence read miss rates in the applications for 4-word cache lines are about 11% for *MP3D*, about 6% for *Pthor*, and 3-1% for the rest of the applications. All applications are run with 24 processors, except for *Pthor* which, due to the large size of the simulations, is run with eight processors only.

3 Analysis of Update Patterns

To address the problem of designing scalable update protocols, we start by devising a new model of how data sharing determines the update patterns. This model, called the *Update Distance* model, is useful because it shows how well optimized update protocols can perform. Older models, like Eggers and Katz's write run model [4], do not provide the right insight for advanced protocols like delayed update protocols. In the following, we first present our model and then apply it to the applications.

3.1 The Update Distance Model

The model is based on the concept of update distance. Given a stream of read and write references by several processors, we focus only on those that are directed to a given word w. A group of consecutive accesses to w by the same processor without any intervening access by another processor is coalesced into a single access: if the group contains only reads, the result is a read; otherwise, the result is a write. Finally, for an access A by a processor in the coalesced trace, the number of intervening writes (not reads) by other processors since the processor last accessed the data is called the update distance of A. Figure 1 shows an example of the update distance for a reference.

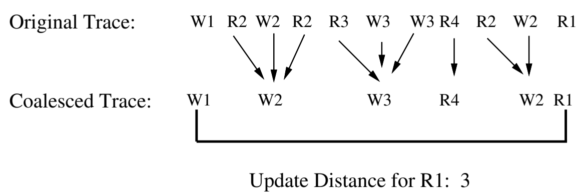

Figure 1: Update distance model. *R1* and *W1* mean that processor 1 issues a read and a write reference respectively. All references access the same word.

The intuition behind the update distance model is to determine how well a competitive update protocol [7] performs in an optimistic scenario. Imagine an ideal scenario where a processor could read and write a variable several times and not update anything until just before a second processor accessed the variable. At that point, the first processor would update all the sharers. In that scenario, the number of updates N that a processor would receive between the issue of two consecutive accesses A and B, separated by accesses from other processors, is given by the update distance for B. Consequently, a competitive update protocol with a threshold equal to N or larger would eliminate a miss in B. In general, to eliminate a miss in an access that has an update distance of N, we would need a competitive threshold equal to at least N. Conversely, a competitive update protocol with a threshold equal to N can only eliminate the misses in accesses that have update distances of N or less.

3.2 Applying the Model to Applications

From the previous discussion, it follows that it is interesting to measure the update distances that real programs exhibit. Therefore, we computed the update distances for each individual variable in our applications and then merged all the data for a given application. Then, we plotted the data as distributions of update distances. To present the distribution, we used the markov matrix format: we show the distribution of the current update distance as a function of the previous update distance for the same variable. Finally, the data that we will present in the plots follows a slightly different definition of update distance: when coalescing references in the original trace to form the coalesced trace (Figure 1), references are never coalesced across a release point. This means that two consecutive writes by the same processor before and after a release point count as two different writes. We do this because the data will be more useful with this change when we try to design real protocols (Section 4). Indeed, real protocols need to enforce special actions at release points.

Figure 2 shows the distributions of update distances for our applications. As indicated before, we use the markov matrix representation of the distributions. In each chart, the two coordinates of the matrix are the previous update distance and the current update distance. Update distances of zero are removed because they do not create cache misses. Also, we have arbitrarily cut the graph at a distance of 25, lumping all distances larger than that in the last bin. The z coordinate gives how frequent a given combination of previous and current update distance occurs. For example, the height of a peak at (*cur_dist*, *prev_dist*) gives the number of times that we have measured an update distance of *cur_dist* that was preceded by an update distance of *prev_dist*.

The figure shows different features for different applications. In *Barnes* and *Pthor*, there is one single dominant peak. The peak is at (1,1), showing that most update distances are one and that they do not change with time. This behavior is caused by the dominant sharing patterns in the applications. For example, *Barnes* repeatedly executes two operations. First, each processor updates a set of bodies, namely those that it owns. Then, in the next time step, all processors read most of the bodies. Obviously, this pattern induces update distances of one. In *Pthor*, there are several data structures that contribute to the main peak. One of them is the per-processor work queues. During execution, when a processor looks for work, it first accesses its own queue. If the queue is empty, it then accesses its neighbor's queue. It often occurs that a queue is alternatively accessed by the owner processor and its neighbor. The resulting accesses to the queue data structure have distances of one.

From this discussion, we see that the (1,1) pattern is easy to generate. Note that an invalidate protocol would suffer as many misses as the height of the (1,1) peak. However, as indicated before, an optimistic competitive update protocol with a threshold of one would eliminate all these misses.

In *Ocean*, there is producer-consumer activity; hence the peak at (1,1). However, *Ocean* also has peaks at (2,2), (3,3), and (4,4). These peaks all result from one type of loop pattern. For example, the (3,3) peak conceptually results from three consecutive doall loops (Figure 3). In a given loop, each iteration writes to one array element. Loop iterations are assigned statically and in the same way in all three loops. After the three loops, all processors read all array elements. Since the loops are separated by release points, the coalesced trace has the same processor writing three times before the other processors read the data. This implies an update distance of three. Obviously, neither a competitive update protocol with a threshold of one nor an invalidate protocol can eliminate the misses in these peaks.

MP3D and *Water* have the typical (1,1) peak, created by several data structures. In addition, however, they also have other shapes generated mostly by migratory data. Migratory data [14] is data that is accessed in sequence by different processors. When a processor accesses the data, it first reads it, and then performs a series of reads and writes

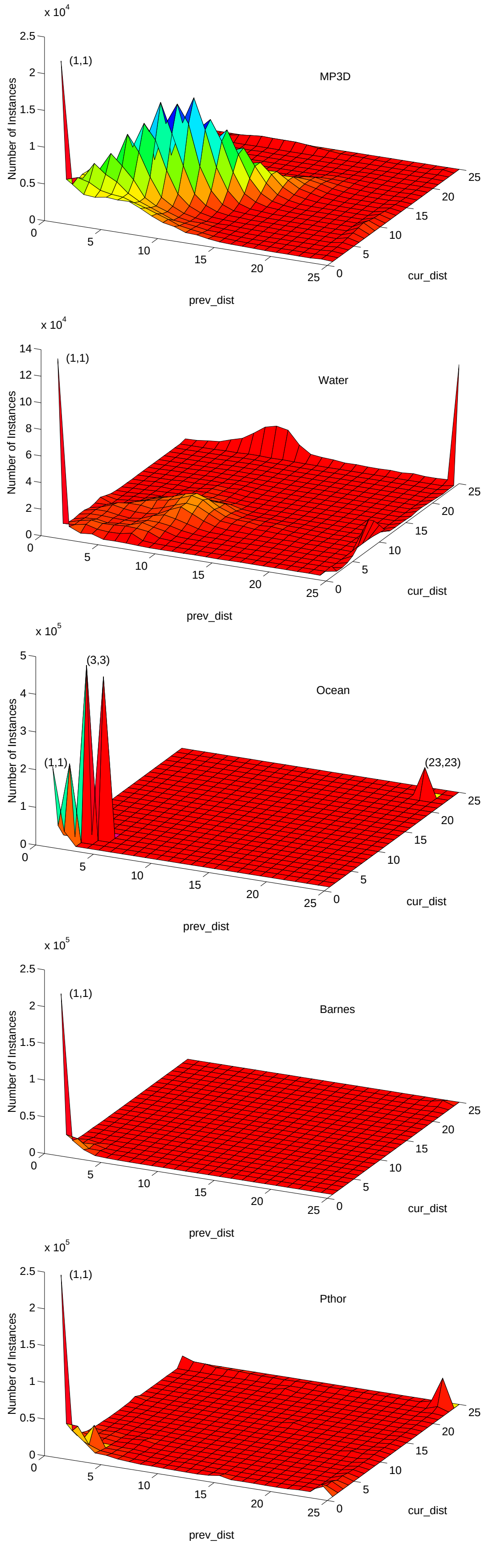

Figure 2: Markov matrix representation of the distribution of update distances. A peak at (x,y) gives the number of instances of an update distance x preceded by an update distance y.

```
DOall  I
    A(I) = ...
End DOall
DOall  I
    A(I) = ...
End DOall
DOall  I
    A(I) = ...
End DOall
// All processors read the whole array A
```

Figure 3: Sharing pattern that causes the (3,3) peak in *Ocean*.

that must include at least one write. If all the processors have similar probability of accessing the migratory data, the update distance pattern will be a bell shape centered in the diagonal region of the matrix. The maximum of the bell will be located at update distances equal to about the number of processors sharing the data.

Interestingly, the applications do not exhibit the expected perfect bell shapes. To understand the reason, recall that *MP3D* simulates a rarefied fluid flow. The space of the fluid is divided into cells. Processors grab and operate on cells when there is a particle crossing it. Cells are shared in a migratory manner by all processors. We can see that the update patterns on the cells certainly create a bell-shaped range in *MP3D*. However, the highest point corresponds to update distances of about 8 or 9. Furthermore, the shape does not have a smooth surface. This suggests that cells are not being shared in a perfectly uniform manner. Possibly, some processors have a higher probability than others to access a given cell. These characteristics will be exploited when we design our update protocol in Section 4.

Similarly, in *Water*, the migratory sharing of a set of water molecules creates unexpected shapes, namely those around (10,10), (25,25), (10,25), and (25,10). These are examples of even less uniform migratory sharing. In *Water*, the access patterns to the N molecules are completely predictable. Indeed, each processor computes the pair-wise interaction between each of the molecules it owns and each of $N/2$ others. In a pair-wise interaction calculation, the two molecules involved are updated. In the process of computing the interactions, each molecule is updated by a deterministic set of processors. These accesses are regular enough to create the four shapes indicated above. However, the order of the accesses to a given molecule is not deterministic enough to create sharp peaks[1]. This is because the updates by the different processors are not performed in lockstep. Instead, they are performed asynchronously with the protection of a lock per molecule. As a result, the actual interleaving of the updates is not fixed and, therefore, the update distances are not clear-cut.

Migratory data sharing patterns are important because invalidate protocols cannot eliminate the misses that such patterns induce. Moreover, a traditional update protocol is unlikely to perform much better: with a very high competitive threshold, it could eliminate the misses; however, the traffic generated is likely to be very high.

To summarize, we note that applications seem to have relatively few different shapes in their matrices. One shape, a peak at (1,1), is generated by several different sharing patterns and is prominent in all applications. A second set of patterns, namely smooth shapes in different regions of the matrix, are often generated by migratory data. Finally, there are other features that are caused by specific sharing patterns, like the sharp peaks in *Ocean*. We need to design update protocols that handle all these cases.

[1]Note that the peak for large *cur_dist* and *prev_dist* in *Water* would transform into a smooth shape if we extended the size of the matrix.

4 Towards Scalable Update Protocols

The update distance model produces a clear picture of the different patterns of data sharing as they appear to optimized update protocols. It therefore gives insight into the features that scalable update protocols need to have. In this section, we use this insight to propose a new family of protocols called distance-adaptive protocols. Before we do that, however, we briefly discuss delayed-competitive protocols, which combine techniques that have already been proposed and which will be used for comparison purposes.

4.1 Delayed Competitive Protocols

These protocols are send-delayed competitive update protocols with thresholds of one and migratory data handling. We generate them by simply combining techniques that have already been proposed by other researchers. We selected their features based on Figure 2.

The first characteristic of these protocols is that they have a threshold of only one. With this threshold, we can eliminate the misses caused by the prominent (1,1) peaks in Figure 2. It is not worth increasing the threshold to capture other shapes: it would result in a small reduction of misses at the expense of much traffic. Unfortunately, Figure 2 depicts an optimistic scenario based on lumping together the references issued by a processor until a second processor accesses the data, or until a release point is reached. This environment, however, can be approximated by using a delayed protocol. Delayed invalidate schemes were proposed by Dubois *et al* [3] and can be trivially extended to update schemes. In these schemes, the updates issued by a processor-cache node are combined in a write cache and propagated to the rest of the system only at release points. This combining reduces the traffic and exposes the behavior that we seek. More importantly, delayed protocols maintain the competitive behavior under multi-word cache lines. For instance, imagine a producer that writes to two words sharing a cache line before the consumer reads any of them: a non-delayed competitive update protocol with a threshold of one would invalidate the line before it is read. Of course, we are assuming that the applications can run under relaxed memory consistency.

Note, however, that in the receiver cache, the update should be processed immediately. Indeed, if the consumer does not immediately acknowledge the update or the invalidation of the line, the competitive behavior may be lost and many update messages may be sent. Therefore, we use send-delayed protocols [3], not receive-delayed.

Finally, since competitive updates do not handle migratory data well, we add a scheme proposed by Nilsson and Stenstrom [9] to detect migratory data in update protocols. In this scheme, once the data is detected as migratory, the directory will supply the data in exclusive mode to the requesting processor and invalidate the processor that used the data before.

4.1.1 Optimizing Migratory Data Handling

Supplying migratory data in exclusive mode saves traffic because the reading processor will not need to request exclusive mode when it writes the data later. Unfortunately, traffic reduction has only a second-order effect on the latency of misses. For this reason, we propose to improve the migratory data handling in [9] by forcing the processor operating on the migratory data to update the home memory at release time. Consequently, when a new processor requests the data, it will get it directly from the home node in 2 hops instead of from a third node in 3 hops. The result is a reduction in the miss penalty. One problem, of course, is that the home memory update is useless if the same processor is the next one to access the migratory data. This, however, is not too likely: the data is likely to migrate to another processor next. Finally, note that this optimization fits well with the send-delayed protocol that we use.

4.2 Distance-Adaptive Protocols

The previous protocols handle at least two major update patterns shown in Figure 2 sub-optimally. First, they cannot handle update distance patterns different from those of producer-consumer and migratory data. For example, they cannot handle the *Ocean* peaks other than (1,1). Second, they still suffer many misses on migratory data. This is in part due to their failure to exploit any determinism that migratory data accesses may exhibit.

To solve these limitations, we propose a new class of update protocols that we call *Distance-Adaptive* protocols. In these protocols, a simple send-delayed update protocol adapts to the sharing patterns of the data structures. The idea is for the directory to record the update distance patterns that the data structures have been subjected to in the past and then use this knowledge to guess which of the processors sharing the data structures should be updated and which ones invalidated. Obviously, we rely on the assumption that past sharing patterns are good predictors of future ones. This is true for all iterative and loop-based algorithms and, indeed, we have yet to find an application which does not exhibit such behavior.

These protocols can be efficiently implemented using a directory with dynamic pointers [11] with some modifications. In the following, we start by describing the most basic protocol in this family, which we call *Base Distance-Adaptive*; then, we examine a more advanced one called *Advanced Distance-Adaptive*; finally we discuss their implementation. For reasons indicated in Section 4.1, all these protocols are send-delayed.

4.2.1 Base Distance-Adaptive Protocol

In this naive protocol, the directory remembers, for each memory line, the last update distance used by each of the processors that have been sharing the line in the recent past. These processors are called *recent sharers*, and are a (hopefully small) superset of the *current sharers* – those that actually have a copy of the line. The directory can then use this last update distance information to invalidate or update processors every time the line is updated in memory.

Conceptually, the directory needs to store, for each of the recent sharers, its previous update distance (*prev_dist*) and its current update distance (*cur_dist*). When the memory receives an update[2] from a processor p, the directory increments *cur_dist* by one for all the recent sharers of the updated line except p. If, as a result, *cur_dist* becomes equal to *prev_dist* for a given processor, we predict that the processor will use the data next. Hence, the directory sends an update to the processor, the processor is marked as current sharer in the directory, and its *cur_dist* is reset to zero. For the others, we predict that they will not use the data. Therefore, if they are current sharers, the directory sends them an invalidation and they are unmarked as current sharers. If, in addition, their *cur_dist* reaches a certain threshold *DistDrop*, they are dropped from the directory altogether. Finally, the state for processor p also changes. Suppose first that p was a recent sharer. In this case, if its *cur_dist* is not zero, the *cur_dist* is stored in *prev_dist* for future use and then reset to zero. If, instead, p was not a recent sharer, a new entry for p is allocated in the directory. Its *cur_dist* is set to zero and its *prev_dist* is set to a large default value for lack of a better one. In both cases, p is marked as a current sharer.

When the memory receives a read request from a processor p, the directory state for p changes in the same manner as in a write. The other sharers, however, are not affected. In particular, the *cur_dist* value of the other sharers remains unchanged since reads do not affect update distances. To see graphically how this protocol works, Figure 4 shows the directory state when two processors share a word with a migratory yet deterministic pattern. After a transient period, all misses are eliminated. In the figure, we use a directory with dynamic pointers, as will be described later.

The guesses of the directory may be wrong in two cases: a recent sharer may access the data before it receives the update (new update distance is smaller than the old one), or a recent sharer may not access the data between the time it is updated and the time it is invalidated (new update distance is larger than the old one). In the first case, the recent sharer will miss. The resulting read or write to memory will be handled as indicated before. It will result in the setting of the sharer's *prev_dist* to the value of the sharer's *cur_dist* and the clearing of the latter.

In the second case, where a recent sharer does not access the data when it is supposed to, we first need to develop a way for the directory to get notified of this. There are different ways for this to happen. In our simple first implementation, the directory gets notified when it receives the acknowledgement to the next message that it sends to the badly-behaved processor. This message will usually be an invalidation. If the acknowledgement says that the processor did not access the data as expected between update and invalidation, the directory simply sets *cur_dist* to *prev_dist*+1 for the processor.

This basic scheme can be optimized in many ways. One optimization to decrease update traffic involves allowing data to become dirty in the caches sooner. In the scheme that we have described, a line can become dirty in the cache of a processor only when all the other recent sharers have been dropped from the directory. Suppose, however, that at a given time, for all the processors currently in the linked list, the difference between *prev_dist* and *cur_dist* is large. Furthermore, suppose that all writes are being performed by one processor at that time. Since no recent sharer is expected to use the data in the next few memory updates, it is inefficient to force the writer processor to update memory. We could invalidate all processors and let the writer keep the data dirty in its cache. This is not currently implemented.

Another optimization concerns a tradeoff between traffic and misses. In the proposed scheme, only one update is sent to each recent sharer, in the hope that the exact distance is correctly guessed. While this approach may work fine for very deterministic update patterns like those in *Ocean*, it may not capture some of the more smooth shapes in Figure 2. One way to handle this is to send several updates to each processor, for example one immediately before, one immediately after, and one at the time of the expected use of the data. This window-based approach, however, causes more traffic. Instead, a better solution is presented in the next protocol.

[2] Because we use a delayed protocol, the frequency of updates is lower than in non-delayed protocols.

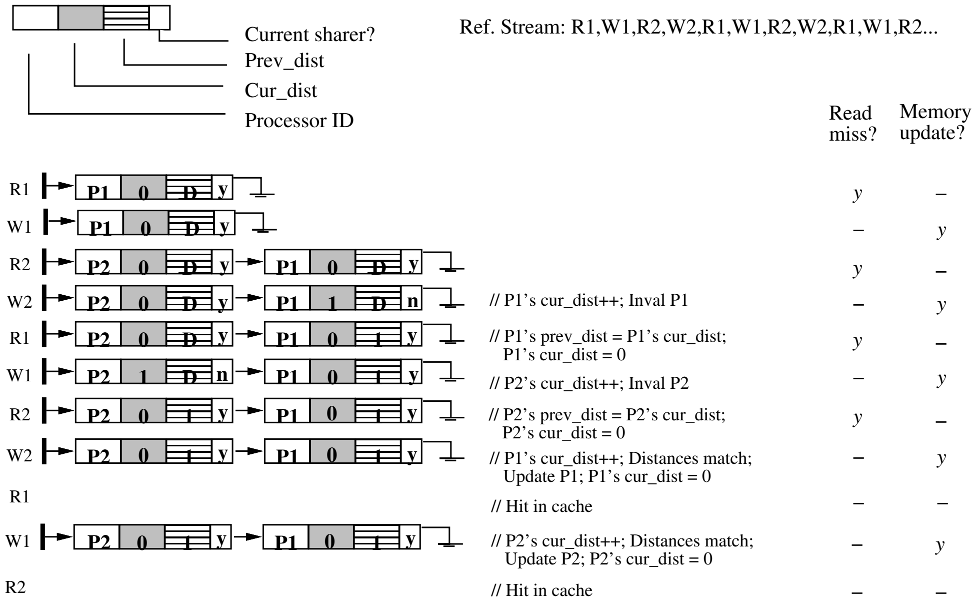

Figure 4: Directory state when two processors share a word with a migratory yet deterministic pattern. The base distance-adaptive protocol eliminates all steady-state misses. In the figure, all references access the same word. *R1* and *W1* represent a read and a write by processor 1 respectively. *D* is the initial value for *prev_dist*.

4.2.2 Advanced Distance-Adaptive Protocol

One shortcoming of the previous protocol is that it only remembers the very recent past for each processor — earlier behavior is forgotten. Furthermore, by assuming that the future will resemble the immediate past, it can only capture the features in the diagonal of the matrices in Figure 2. To capture the complete sharing behavior of the data, we propose to store the whole markov matrix in the directory and use it to generate educated guesses of future update distances. This protocol we call *Advanced Distance-Adaptive*.

In this scheme, like in the base distance-adaptive protocol, when a memory module receives an update from a processor p, the directory traverses the list of recent sharers for the updated memory line. Now, however, for each recent sharer except p, we increment *cur_dist* and use the pair (*prev_dist*,*cur_dist*) to access the matrix. The count that we read from the matrix entry is a measure of how useful updating the processor is likely to be. If the count is higher than a certain threshold stored in the *ThresholdReg* register, the directory sends an update to the processor; otherwise, if the processor is a current sharer, an invalidation is sent. In all cases, the current sharer status is updated as necessary. Clearly, with this scheme, we weigh in all past update distances and, in addition, capture all shapes in the matrix, not just those in the diagonal. Finally, the state of the writer processor p changes like it did in the base distance-adaptive protocol. Before saving *cur_dist* in *prev_dist* and clearing *cur_dist*, however, we update the markov matrix. Indeed, to incorporate in the markov matrix the new update distance just recorded, we increment by one the count in location (*prev_dist*,*cur_dist*).

Similarly, when the memory receives a read request from a processor p, the directory state changes as in the base distance-adaptive protocol. The only exception is that location (*prev_dist*,*cur_dist*) of the markov matrix is incremented by one to record the event.

The markov matrix should also be updated when the directory sends an update to a sharer. This is because the directory is guessing that the data will be accessed and, therefore, a new update distance needs to be recorded. Unfortunately, the guess may be wrong. In the base distance-adaptive protocol, the directory assumed that the guess would be right, made the necessary directory state changes and, when the directory received a message informing it that the data had not been used, the directory undid the directory state changes. In the advanced distance-adaptive protocol, however, we do the opposite: after sending an update, the directory state does not change until a confirmation that the updated processor has used the data is received. As indicated in Section 4.2.1, the confirmation can be included in the acknowledgement to the next message sent to the processor. When the confirmation is received by the directory, we add one to location (*prev_dist*,*cur_dist*-1) in the matrix, copy *cur_dist*-1 to *prev_dist*, and set *cur_dist* to 1. This 1 factor appears everywhere because the directory receives the confirmation one write late.

Clearly, the matrix is being built on the fly while it is being used. By doing so and regularly pruning the matrix, we can capture changes in the regime of the program. In addition, when a context switch occurs, the matrix for the old process is saved in memory and the matrix for the new process is restored. Note that the size of the matrix is only $DistDrop*DistDrop$ integers. Since $DistDrop$ is 15 in the experiments of the evaluation section, the matrix size is less than 1 Kbyte. The evidence from Figure 2 suggests that, except in *Water*, most of the shapes occur for small distances. Furthermore, we feel that long distances in *Water* could be eliminated by allowing data to become dirty in the caches sooner, as we explained in Section 4.2.1. Consequently, we expect the size of the interesting part of the matrix to increase more slowly than the number of processors.

We are able to keep the storage overhead low because a single matrix services requests from all the processors and accumulates information for all the data structures in the corresponding memory module. This has proved not to be a problem. In particular, there is no interference among the different data structures that share the matrix. This can be explained by looking at Figure 2: there is little interference because the matrices have few different shapes. This, in turn, is due to the use of a delayed protocol: many data structures end up displaying the same behavior and appearing in the (1,1) entry. From the figure, therefore, we conclude that it is unlikely that our scheme will send many misdirected updates or invalidates as a result of getting confused by different data structures.

The threshold count that is used to decide whether or not to send an update message can be generated in different ways. A very simple strategy is to use the sum of all the counts in the matrix shifted right by a constant. This strategy only requires incrementing a global counter every time an entry in the matrix is incremented. With this scheme, we can even control how aggressive the protocol is. This is done by varying the constant. If this constant is large, much update traffic can be generated but fewer misses will occur. The opposite occurs if the constant is small. We could even use the degree of network utilization as a feedback to guide the setting of the constant dynamically.

A second, more complex scheme is to send updates only if we are in any of the n highest peaks of the matrix. In this case, the *ThresholdReg* register contains the count in the lowest of all these peaks. The coordinates of the chosen peaks could be kept in registers for fast access, in case another, higher peak appears and we want to replace the lowest of the old ones with the new one. Obviously, this approach requires some bookkeeping to keep all the registers up to date. Finally, a more accurate yet complex scheme is to send the update or not based on the probability that the processor will use the data. This probability can be estimated by dividing the count read from the matrix entry by the sum of all the counts in the same *prev_dist* row of the matrix.

4.2.3 Implementation

Distance-adaptive protocols can be implemented efficiently. In this section, we describe an implementation of a directory that is relatively simple, small, and processes coherence transactions fast. The complete state machine in the directory is shown in Figure 5-(a). It is composed of three modules connected by a control pipeline: directory module, markov module, and message module. The directory module contains the directory, which is slightly modified to support distance-adaptive protocols. The markov module, which is required only for the advanced distance-adaptive protocol, contains the markov matrix and the logic necessary to manipulate it. Finally, the message module builds and decodes messages and is not specific to distance-adaptive protocols. In the following, we describe the first two modules.

Directory Module

We support distance-adaptive protocols by using a slightly-modified directory with dynamic pointers. Dynamic pointers have been proposed by Simoni and Horowitz [11] and implemented in FLASH [8]. In a dynamic pointer scheme, each link in a linked-list stores the ID of the sharer processor. As shown in Figure 4, we modify the implementation so that each link additionally stores the current and previous update distances for the processor, and whether or not the processor is a current sharer. Recall that, while all processors in the list are recent sharers, not all are necessarily current sharers. We use 1 bit for the current sharer field and 4 bits for each of the distances. In the following, we analyze the resulting storage overhead and any slowdown caused by the use of distance-adaptive protocols.

The addition of these fields does not increase the directory storage much. For instance, assume the implementation described by Simoni and Horowitz [11] and that each processor node has 32 Mbytes of main memory. We run our applications under an invalidate, competitive update, and the two distance-adaptive protocols. We select the application that requires the largest number of links. The number of links required by each protocol after rounding up to the next power of two is 32 K, 64 K, 128 K, and 128 K for invalidate, competitive update, base distance-adaptive protocol, and advanced distance-adaptive protocol respectively. Table 2 shows the percentage of memory taken by the directory for several different cache line sizes and protocols. From the table we see that, while distance-adaptive protocols have a higher overhead, the difference between all the protocols is small, especially for long lines. Therefore, we conclude that the storage overhead of the links is no more of a problem for distance-adaptive protocols than it is for the other protocols.

Table 2: Percentage of memory taken by directory linked lists for different protocols and cache line sizes. The figures are rounded up to the next power of two.

Line Size (Words)	Invalidate (%)	Compet. Update (%)	Base Distance Adap (%)	Advanced Distance Adap (%)
4	11.9	13.0	14.7	14.7
8	5.5	6.1	6.9	6.9

Similarly, the extra directory operations required in distance-adaptive protocols slow down directory accesses little. First, any increments and comparisons of distances are done off the critical path of any processor read access. Furthermore, in the case of writes, any such operations are performed in a pipelined fashion, thereby keeping overhead to a minimum. Recall that, in a write, the whole list is traversed. In the base distance-adaptive protocol, for each link in the list except maybe one, *cur_dist* is incremented and compared to *prev_dist*, and an update or invalidate message may be sent. These operations are pipelined according to Figure 5-(b). In the *fetch* stage, both distances are fetched. In *cmp/up*, we compare *cur_dist+1* to *prev_dist* and update *cur_dist* and possibly *prev_dist*. Finally, a message may be built in stage *msg*. A dynamic pointer implementation of an invalidate protocol can be easily built with a two-stage pipeline: *fetch* and *msg*. Consequently, given a linked list, the distance-adaptive protocol needs only one cycle more than the invalidate protocol to traverse the list. The figure also shows that the pipeline for the advanced distance-adaptive protocol is similar. This pipeline will be discussed later.

Unfortunately, the linked lists in distance-adaptive protocols tend to be longer than in invalidate or competitive update protocols. This is because they keep more than simply the current sharers. For the applications where migratory sharing dominates (*MP3D* and *Water*), the lists in distance-adaptive protocols are, on average, between 3 and 4 times longer than in invalidate protocols (hence the 128 K and 32 K links required by each protocol after rounding up to the next power of two). For the other applications, however, which have more producer-consumer patterns, the length of the linked-lists does not change much with the protocol. Clearly, as the length of the linked lists increases, the occupancy of the directory increases proportionally. Overall, however, this higher occupancy in distance-adaptive protocols does not slow down requests significantly. On average, only 2% of the read latency in base and advanced distance-adaptive protocols is due to the directory being busy processing another request. Overall, therefore, the higher directory complexity for these protocols does not affect the directory speed much.

Markov Module

The markov module (Figure 6) contains the markov matrix and the logic necessary to manipulate it. The whole matrix is small enough (about 1 Kbyte) to be stored in a

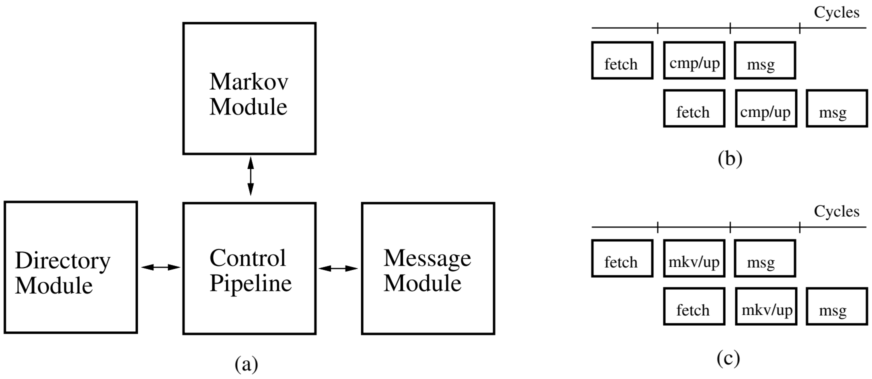

Figure 5: Directory (Chart (a)) and pipeline for the base distance-adaptive (Chart (b)) and advanced distance-adaptive (Chart (c)) protocols.

fast SRAM. This memory is addressed with an 8-bit address formed by the *prev_dist* and *cur_dist* parameters. The *sum* register contains the sum of all the counts in the matrix and is incremented every time an entry in the matrix is incremented. To perform these increment operations we need two 32-bit incrementers. Furthermore, to decide whether to send an update or an invalidation, we need a comparator to compare the count in the matrix entry to a threshold count found by dividing the sum register by a power of two. This division is easily implemented by shifting the sum register right a given number of bits. In Figure 6, we accomplish this shift by taking only the 29 most significant bits of the sum register. Finally, to adapt to changing sharing patterns, we regularly prune the matrix. This is done by regularly shifting right the sum register and all entries in the matrix a few bits.

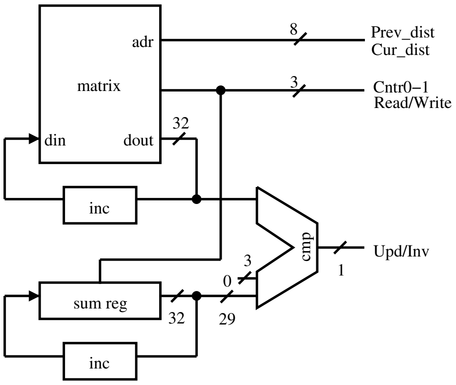

Figure 6: Design of the markov module. By using only the 29 most-significant bits of the sum register, we effectively divide the sum register by 8.

The pipeline for the advanced distance-adaptive protocol (Figure 5-(c)) is similar to the one for the base distance-adaptive protocol. In the advanced protocol, the *cmp/up* stage has been renamed *mkv/up*. In this stage, we read the matrix and update the matrix (sometimes), *cur_dist*, and *prev_dist* (sometimes). The operations performed on the matrix are as follows. If the transaction is a write and the link being processed does not point to the processor originating the transaction, location Matrix[*prev_dist*,*cur_dist*+1] is compared to the shifted sum register and the Upd/Inv output asserted. If the transaction is a write or a read and the link being processed points to the originating processor, location Matrix[*prev_dist*,*cur_dist*] and the sum register are incremented. Finally, if this is an acknowledgement of a previous update, location Matrix[*prev_dist*,*cur_dist*-1] and the sum register are incremented. These operations are controlled by the Read/Write and two control signals, and are performed in the *mkv/up* stage. Therefore, the time required to service a memory access in the advanced distance-adaptive protocol will be the same as in the base distance-adaptive protocol.

5 Evaluation

In this section, we compare the performance of the proposed protocols. As a reference, we also evaluate state of the art non-delayed invalidate and update protocols. The protocols that we examine are:

Protocols used as a reference:

- *Inval+Mg*: Non-delayed invalidate *with migratory data handling* as described in [13]. We call this protocol base invalidate protocol. This protocol is quite sophisticated.
- *CompUp*: Non-delayed competitive update with threshold of one. We call it base update protocol.

Delayed competitive update protocols:

- *DCU*: Delayed competitive update with a threshold of one.
- *DCU+Mg*: *DCU* with Nilsson's migratory data handling [9].
- *DCU+Mg+Hm*: *DCU+Mg* plus the home memory update optimization for migratory data that we propose in Section 4.1.1.

Distance-adaptive protocols. We set *DistDrop* and the initial value of *prev_dist* to 15:

- *BasDAdap*: Base distance-adaptive.
- *AdvDAdap*: Advanced distance-adaptive generating a matrix for 4- or 8-word cache lines at run-time. The threshold count used to decide whether or not to send an update is 5% of the sum of all the counts in the matrix.

In the following, we first compare the read misses in the secondary cache and the traffic induced by the different protocols. Then, we examine the execution time of the applications. In our discussion, we will often point to features in Figure 2, even though the figure corresponds to caches with single-word lines and the simulations in this section were performed with 4- and 8-word lines. While the markov matrices vary somewhat with the cache line size, the changes do not justify plotting the matrices again: most of the features remain in the same place. The most noticeable change is that a small to medium-sized mountain-like shape appears for positions with large values of both *cur_dist* and *prev_dist*, in one corner of the matrix. This feature is largely caused by false sharing. Indeed, it appears because, before a processor reuses a given cache line, several other processors write to other words in the same cache line, increasing the first processor's update distance.

5.1 Cache Misses and Traffic

The number of read misses in the secondary caches and the amount of network traffic for the seven different protocols is shown in Figures 7 and 8 respectively. Each figure shows data for caches with 4-word lines (leftmost set of seven bars for each application) and 8-word lines (rightmost set of bars). For each application, the misses are normalized to the number of misses in the base invalidate protocol for 4-word cache lines, while the traffic is normalized to the number of words transferred in the base update protocol for 4-word cache lines. In the miss charts, we separate misses on migratory data from the rest, while in the traffic charts, we divide the traffic into words transferred in data-carrying messages and non data-carrying messages. The former refer to cache lines and updates, while the latter include invalidations and acknowledgements to both invalidations and updates. Finally, for each bar in the traffic charts, we show the average number of hops traveled by a message in a read miss to shared data. This number ranges from two, when the data is always supplied by the home node, to three when it has to be supplied by a third node. Obviously, the larger this number is, the longer the requesting processor will be stalled in a read miss.

It is well-known that the sharing patterns of an application influence the number of misses induced by a protocol. For this reason, we recall from Table 1 the dominant sharing patterns of the applications: migratory data patterns in *MP3D* and *Water*, producer-consumer patterns in *Ocean*, and a mix of several patterns in *Barnes* and *Pthor*.

We focus first on the two applications where migratory data dominates. Recall that there is little that competitive update protocols can do to optimize migratory data: either the resulting traffic is high or the number of misses is high. Our three schemes (*DCU*, *DCU+Mg*, *DCU+Mg+Hm*) try to avoid creating too much traffic. The result is that, for example, *DCU* has fewer misses than the base invalidate protocol (7% fewer for *MP3D* and 13% for *Water*) with a relatively modest traffic increase (4 times more for *MP3D* and *Water*). We feel that this makes *DCU* an interesting protocol. The reason why the misses decrease this much with this tolerable traffic increase is that, by using a delayed protocol, many variables now appear in the (1,1) peak of the matrix in Figure 2 and, therefore, behave as producer-consumer variables.

The three delayed competitive protocols represent different choices. In *DCU+Mg*, when a processor accesses migratory data, the old owner is invalidated. If this old owner accesses the data next, it will miss. In *DCU*, the old owner would have been updated and therefore would not miss the second time. Thus, *DCU+Mg* induces less update traffic but more misses than *DCU* (6% more in *MP3D* and 9% more in *Water*). In *DCU+Mg+Hm*, we still invalidate the old owner but the home memory is up-to-date. Therefore, we will have the same misses as *DCU+Mg* but each miss is cheaper. Indeed, the data is now fetched from the home in two hops instead of from another processor in three hops. This can be seen from the hop numbers in Figure 8: approximately three in *DCU+Mg* and two in *DCU+Mg+Hm*. This means

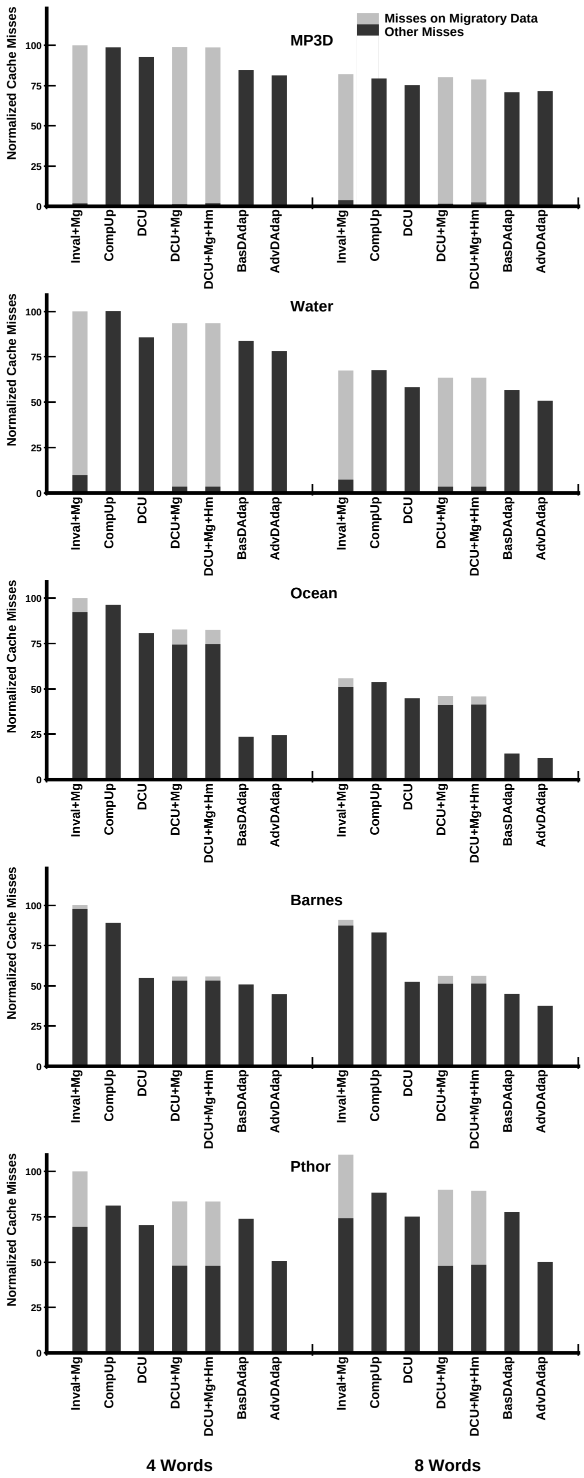

Figure 7: Read misses in the secondary caches for different protocols and cache line sizes. For a given application, the number of misses is normalized to the base invalidate protocol for 4-word cache lines.

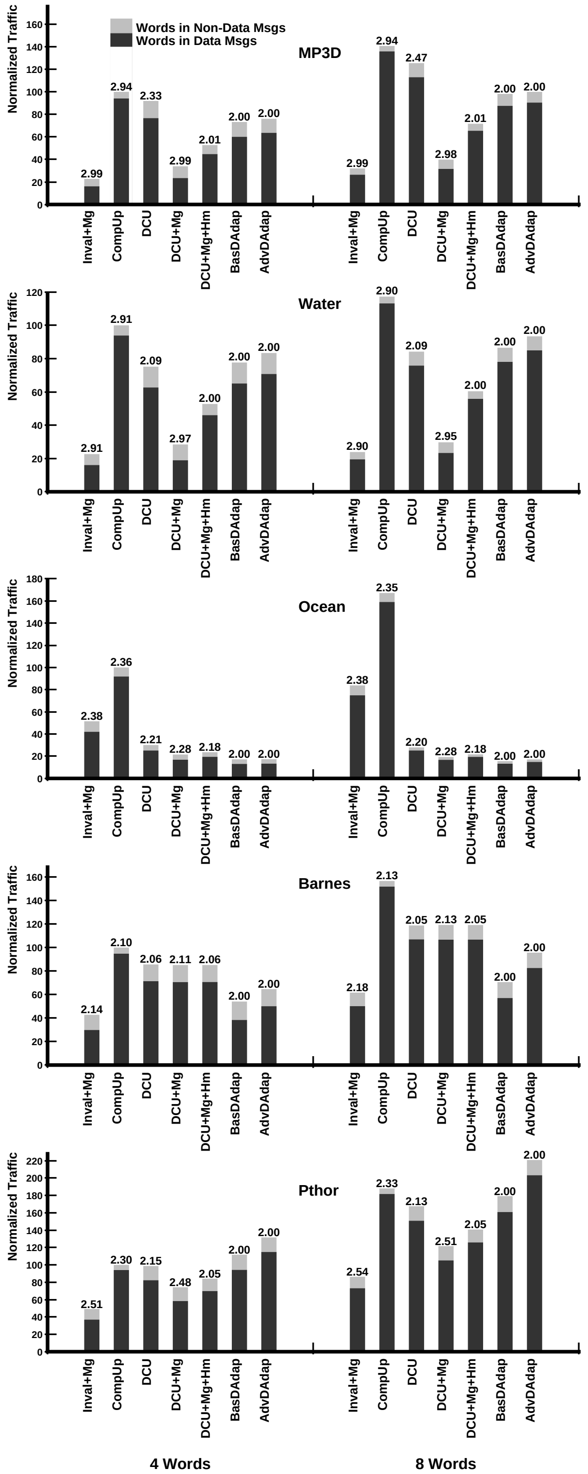

Figure 8: Network traffic in words for different protocols and cache line sizes. For a given application, the amount of traffic is normalized to the base update protocol for 4-word cache lines. The numbers on top of the bars are the average number of hops that a message travels in a read miss to shared data.

that each miss is about one third cheaper. Overall, however, we need to perform simulations to determine the relative performance of *DCU*, *DCU+Mg*, and *DCU+Mg+Hm* for a given network bandwidth and latency.

For the distance-adaptive protocols, we make three observations. First, they have fewer misses than the *DCU* protocol. On average, the reduction is 12% for *MP3D* and 8% for *Water* (Figure 7). Part of the reason for the reduction is that these protocols capture some of the peaks generated by migratory data in Figure 2 outside the (1,1) coordinate. They can capture these peaks because the sharing of migratory data is not completely random in these applications. The protocols will try to discover if there is a certain pattern in the sharing. In general, the less random the sharing pattern is, the higher the effectiveness of the protocols is. In Figure 2, the features in *MP3D*'s matrix are less random than the ones in *Water*'s. Note that part of the peak for large distances in *Water*'s matrix is actually a bell shape spread over some distance. The second observation is that the updates sent by these protocols are more successful than those sent by *DCU*. This can be deduced from the relatively low traffic caused by these protocols (Figure 8). Indeed, the traffic is 20% lower than in *DCU* for *MP3D*, and approximately the same for *Water*. Finally, we note that, predictably, *BasDAdap* and *AdvDAdap* have a similar number of misses for *MP3D*, while *AdvDAdap* has fewer misses for *Water*. The reason is that most of the features in *MP3D*'s matrix are in the diagonal (Figure 2), while this is not the case for *Water*. Since *BasDAdap* can only capture the diagonal, it is less effective in *Water*.

We now focus on the rest of the applications. As expected, the delayed competitive update protocols generally eliminate many misses (Figure 7). In *Ocean* and *Barnes*, the three protocols have about the same effectiveness, removing 20% and 45% of the misses in the invalidate protocol respectively. In *Pthor*, *DCU* removes about 30% of the misses, while *DCU+Mg* and *DCU+Mg+Hm* are a bit less effective. The reason is that *Pthor* has some migratory data and, therefore, the latter two protocols try to keep the traffic lower (Figure 8) at the expense of some misses. This migratory data creates some small features in *Pthor*'s matrix for large current or previous update distances (Figure 2). We note in passing that the base update protocol *CompUp* performs poorly in two of these three well-behaved applications. The reason is that, as indicated in Section 4.1, non-delayed competitive update protocols can become ineffective for multi-word cache lines: updates to multiple words of the same line before the words are used causes the invalidation of the line.

The distance-adaptive protocols further reduce the number of misses. We focus first on *Barnes* and *Pthor*. The simplest scheme, *BasDAdap*, cannot eliminate many more misses than *DCU* because the dominant peak in the diagonal of the matrices in Figure 2 is in (1,1). Such a peak is already captured by *DCU*. For this reason, *BasDAdap* removes only 7% more misses than *DCU* in *Barnes*. For *Pthor*, there is practically no change. For *AdvDAdap*, the miss reductions relative to *DCU* are larger: 18% and 28% in *Barnes* and *Pthor* respectively. The reason is that the protocol now captures the peaks outside the diagonal of the matrices. For these two applications, especially for 4- and 8-word cache lines, the matrices have some small peaks around the (1,1) location and mountain-like shapes for large current and previous update distances. Note that misses in distance-adaptive protocols tend to be cheap: according to Figure 8, the number of hops is always two, as opposed to around 2.15 for *DCU*. This may compensate for the higher traffic that the protocol induces in *Pthor*.

Finally, *Ocean* is an example of the effectiveness of distance-adaptive protocols (Figure 7). Delayed competitive update protocols can only capture the (1,1) peak in the *Ocean* chart of Figure 2. Distance-adaptive protocols, however, can capture the remaining peaks. In fact, since all the remaining peaks are in the diagonal, both the base and advanced protocols capture them. As a result, both protocols reduce the number of misses in the delayed competitive protocols by about 70%. There are few remaining misses. Note that this encouraging result is accomplished without increasing the amount of traffic (Figure 8).

Overall, we conclude that distance adaptive protocols, particularly *AdvDAdap*, are effective in eliminating misses without creating excessive traffic.

5.2 Execution Time

The previous changes in misses and traffic affect the execution time of the applications. Figure 9 shows the execution time for the different protocols and cache line sizes. This figure is organized like the previous ones except that we dropped *CompUp* because of its bad performance. In the figure, the bars are normalized to the base invalidate protocol for 4-word cache lines. The execution time is divided into time that the processor spends executing (*Busy*), performing synchronization or waiting because the write buffer or the write cache are full (*Write and Sync*), and stalled on read misses (*Read*).

From the figure, we see that for *Water* and *Barnes*, most of the time is spent executing or synchronizing. Therefore, the choice of what protocol to use will have relatively little impact. In the other three applications, however, the *Read* category is larger and, therefore, the impact of the protocol will be clearer.

Examining first the three delayed competitive update protocols, we see that, most of the time, they are a bit faster than the base invalidate protocol. Recall that the invalidate protocol has support for migratory data and, therefore, is quite sophisticated. However, not all of the competitive protocols perform equally. *DCU+Mg* tends to perform the worst because it favors traffic reduction at the expense of misses, and misses are very expensive. *DCU* performs reasonably well in *MP3D* and *Ocean* but not in *Pthor*. This is due to *Pthor*'s many tiny critical sections, where the updates issued by the processor have to be propagated to other processors before lock releases. This slows down the release and the ensuing acquire by another processor. Therefore, the *Write and Sync* time is high. In *DCU+Mg+Hm*, the data is propagated only to the home memory and, therefore, *Write and Sync* is smaller. Overall, the most cost-effective delayed update protocol is probably *DCU* which, over the 5 applications, decreases the execution time by 5% relative to the invalidate protocol. If higher performance is required, *DCU+Mg+Hm* decreases the execution time by 8% at a higher cost.

We now consider the distance-adaptive protocols. These protocols perform well in all applications where the *Read* time matters: *MP3D*, *Ocean*, and *Pthor*. For these three applications, the distance-adaptive protocols reduce the execution time by 20% relative to the invalidate protocol and 13% relative to the best delayed competitive one (*DCU+Mg+Hm*). The reason why the gains are relatively small in *Pthor* was indicated before: the delay introduced by the updates slows down the passing of synchronization variables among processors. As a result, the *Write and Sync* time is high. This time can be reduced by letting processors keep dirty data sooner and not update memory (Section 4.2.1). The new protocols also reduce the read stall in *Barnes* and *Water*. The impact, however, is negligible because the original read stall time for these two applications

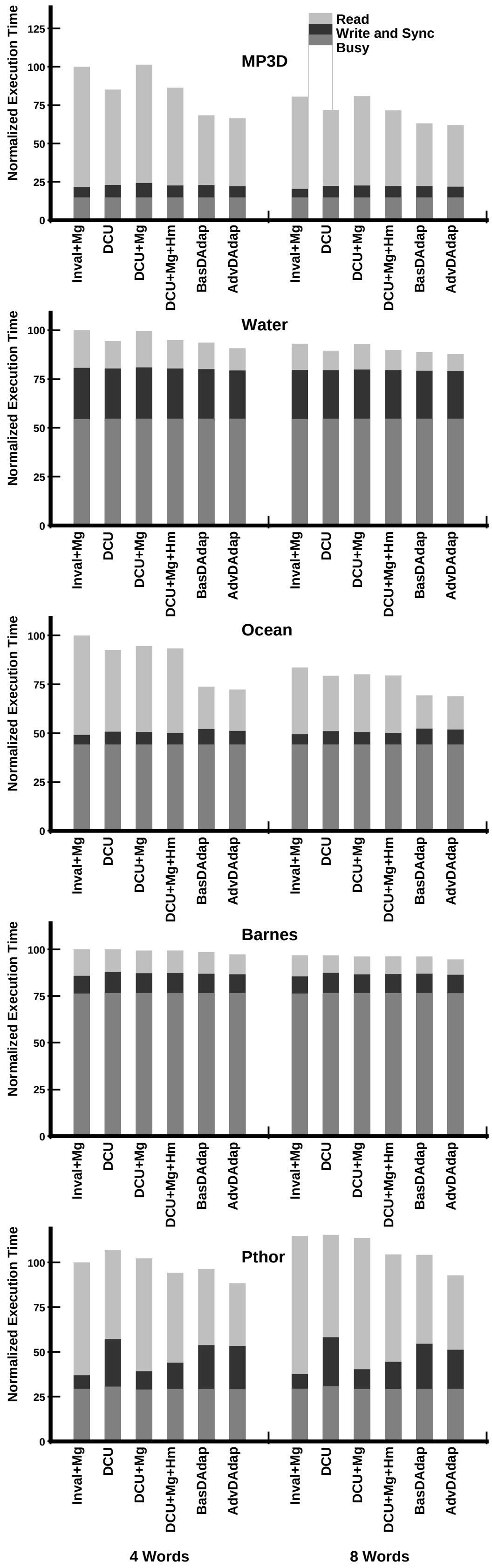

Figure 9: Execution time for different protocols and cache line sizes. For a given application, the execution time is normalized to the base invalidate protocol with 4-word lines.

is small. Overall, given that we have evaluated only a primitive design of distance-adaptive protocols, we feel that these protocols are promising.

If we compare the *BasDAdap* and *AdvDAdap* protocols, we see that they perform comparably except for *Pthor*. The reason is that, of all the applications with a considerable *Read* time, only *Pthor* has a markov matrix for 4- and 8-word cache lines that has important features outside the diagonal. In *Pthor*, *AdvDAdap* is 10% faster than *BasDAdap*. While it is true that *AdvDAdap* is more complicated to implement than *BasDAdap*, we feel that it is better to use *AdvDAdap* in high performance systems. The reason is that there is evidence from Figure 2 that features outside the diagonal of the matrices are common. Supporting only features in the diagonal is likely to make the protocol too unstable. Overall, *AdvDAdap* decreases the execution time of the 5 applications by 15% and 10% on average over efficient invalidate and delayed competitive-update protocols respectively.

6 Concluding Remarks and Future Work

This paper makes two contributions. First, it presents a model of sharing called the Update Distance model that gives insight into the kinds of update patterns that optimized update protocols need to handle. Second, it uses this model to design a new family of scalable update protocols called Distance-Adaptive Protocols. We presented an efficient implementation of the protocols using dynamic pointers. Furthermore, we compared the new protocols against efficient update and invalidate protocols, including a set of delayed competitive update protocols. The result is that distance-adaptive protocols outperform the others in all applications. The advanced distance-adaptive scheme decreases the execution time of 5 applications by an average of 15% and 10% over efficient invalidate and delayed competitive-update protocols respectively.

While the results are encouraging, we have only presented a primitive version of the new protocols. Many refinements of these protocols are possible and their exploration can lead to better designs. One such refinement is to use the state stored by these protocols in the directory to perform adaptive sequential prefetching. This scheme involves testing the correlation between consecutive memory lines based on their update distances. In a naive implementation, when a directory receives a read request for line l from processor p, the directory supplies l and then examines the directory state for consecutive lines with increasing addresses. These lines are also sent to p if their *cur_dist* for p is the same as that in l. The prefetching stops at the first line with a different *cur_dist* for p. Preliminary results show that these prefetches eliminate many misses remaining in the *AdvDAdap* protocol. Furthermore, they do so with high efficiency. This means that extra traffic is not a major concern. Another issue that we are studying is distance-adaptive protocols for very long cache lines. We are currently addressing these and other issues.

Acknowledgments

We thank Per Stenstrom, the referees, and the graduate students in the I-ACOMA research group for their feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] F. Dahlgren and P. Stenstrom. Reducing the Write Traffic for a Hybrid Cache Protocol. *Journal of Parallel and Distributed Computing*, 26(2):193–210, April 1995.

[3] M. Dubois, L. Barroso, J.C. Wang, and Y.S. Chen. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *Supercomputing 91*, pages 197–206, November 1991.

[4] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–382, May 1988.

[5] D. Glasco, B. Delagi, and M. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the 27st Annual Hawaii International Conference of System Sciences*, pages 543–545, January 1994.

[6] S. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[7] A.R. Karlin, M.S. Manasee, L. Rudolph, and D.D. Sleator. Competitive Snoopy Caching. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 244–254, 1986.

[8] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[9] H. Nilsson and P. Stenstrom. An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic. In *Proceedings of Parallel Architectures and Languages Europe (PARLE'94)*, pages 363–374, June 1994.

[10] H. Nilsson, P. Stenstrom, and M. Dubois. Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models. In *Future Generation Computer Systems*, pages 247–271, June 1995.

[11] R. Simoni and M. Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[12] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[13] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[14] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.