Improving the Performance of Software Distributed Shared Memory with Speculation

Department of Computer Sciences Technical Report 2002-57

Michael Kistler[†*]

[†]IBM Austin Research Laboratory
Austin, TX 78758
http://www.research.ibm.com/arl

Lorenzo Alvisi[*]

[*]Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
http://www.cs.utexas.edu/

Abstract

We study the performance benefits of speculation in a release consistent software distributed shared memory system. We propose a new protocol, Speculative Home-based Release Consistency, that speculatively updates data at remote nodes to reduce the latency of remote memory accesses. Our protocol employs a predictor that uses patterns in past accesses to shared memory to predict future accesses. We have implemented our protocol in a software distributed shared memory system that runs on commodity hardware. We evaluate our protocol implementation on a number of software distributed shared memory benchmarks and show that it can result in significant performance improvements.

1 Introduction

A distributed shared memory (DSM) system allows a collection of computers (nodes), connected by a high-speed network, to be used as a single computing resource. Applications can use the familiar shared-memory programming model but still benefit from the additional processing power available in the system. To provide the illusion of a shared memory, the DSM system intercepts accesses to data that physically resides in the memory of a remote node and executes a *consistency protocol* to bring the data to the local node for processing. To improve performance, DSMs commonly cache data from remote nodes in local memory. The consistency protocol ensures that all copies of the data remain consistent even though they may be accessed concurrently across multiple nodes of the system. A *memory consistency model* specifies the behavior of memory as seen by the application program.

It restricts the set of values that may be returned from the memory system at each point in the execution of a program. The memory consistency model typically assumed is *sequential consistency*, which requires that a read operation return the value most recently written, according to some total ordering of memory operations that is consistent with the program order of each of the nodes [14].

DSM systems can be implemented in hardware, software, or as a hybrid hardware-software system. Software DSM systems are attractive because they can be implemented using industry standard hardware. Unfortunately, these systems typically fail to provide performance comparable to either hardware DSM systems or to message-passing systems [6, 15]. Numerous approaches have been developed to improve the performance of software DSM systems, most notably the use of memory consistency models that relax the requirement of a total ordering of memory accesses in sequential consistency. One of the most popular of these relaxed models is *release consistency*, which leverages synchronization operations already present in a correct shared-memory parallel program to create a partial ordering of memory operations. Many protocols have been developed to implement the release consistency memory model, but *lazy release consistency (LRC)* [2] protocols generally achieve the best performance for typical DSM-style applications. LRC protocols defer sending updated data until it has been explicitly requested by other nodes, which greatly reduces data traffic. While LRC protocols do achieve superior performance in comparison to other software DSM systems, further improvements are still required to make these systems competitive.

In this paper, we explore whether *speculation* can be used to narrow this performance gap. Speculation can improve performance by weakening dependencies in program executions. In particular, it allows critical path processing to proceed in parallel with subordinate processing on which it depends. Speculation can take one of two forms: 1) predict the outcome of subordinate processing when the critical path requires this outcome and then verify this prediction in parallel with continued critical path processing, or 2) predict which subordinate processing will be required by the critical path and then perform this processing ahead of the point where its outcome is required. Speculation has been applied in the past to a variety of contexts in computer hardware and software, including branch prediction [21], value prediction [19], and data prefetching [5, 16].

The focus of our work is to study how speculation can be used to improve the performance of a software DSM system. Since the primary source of latency in these systems is the time required to obtain a copy of remote data for access by the local processor, we attempt to identify which remote data will be required by the application. We then perform the protocol actions to transfer this remote data before the application attempts to access it. Thus, our approach is an instance of the second form of speculation.

In this paper we make two main contributions. First, we present a new consistency protocol for software DSM

called Speculative Home-based Release Consistency (SHRC), which speculatively updates data at remote nodes based on predictions of future memory accesses made by a *memory sharing predictor* [12]. Second, we describe an implementation of our protocol and report on its performance for a suite of eight benchmark DSM applications. We discover that speculation can reduce the execution time of applications with regular access patterns by 40% to 50%. For applications with less regular access patterns, our speculative protocol can still achieve significant performance improvements of up to 30%. Not surprisingly, applications with irregular access patterns do not gain significantly from speculation and may even experience performance degradations. The substantial performance gains that can result from speculation suggest that understanding whether an application is amenable to speculation is well worth the effort.

The rest of this paper is organized as follows. Section 2 provides a brief overview of home-based software DSM protocols, and Section 3 describes the Speculative Home-based Release Consistency protocol. Section 5 presents performance results for a prototype implementation of the SHRC protocol on a suite of benchmark programs. Section 6 summarizes the related work, and Section 7 concludes the paper.

2 Background

Our speculative protocol is a *home-based* software DSM protocol. Most software DSM protocols use the virtual memory management (VMM) facilities in the hardware to trap memory accesses, and thus uses a VMM page as the unit of sharing. Software DSM systems also typically implement a relaxed memory consistency model, such as release consistency. In a home-based protocol, each page of shared memory is assigned a *home node* that is responsible for maintaining and distributing the data stored on the page to other nodes of the system. In this section we describe the features of home-based software DSM protocols that are important for our purposes—a comprehensive discussion can be found in Iftode's dissertation [8].

Program operation on each node is divided into intervals delimited by synchronization operations, such as locks and barriers. All synchronization operations are classified as either an *acquire* or a *release*. Lock acquire and release operations are classified in the obvious manner. A barrier operation is classified as a release followed by an acquire. Each node maintains a *logical clock* [13] which is incremented whenever the node issues a synchronization operation.

For each shared page, the protocol maintains at each node a *vector time stamp* which contains an entry for each of the nodes in the system. The vector timestamp maintained by node r for page p specifies the version of the page that r must access in order to satisfy the memory consistency model. In particular, if t is the value of the i-th entry of r's vector timestamp for p, then r should access a version of p that contains all updates that occurred on node i

before i incremented its logical clock to $t + 1$.

In a home-based protocol, the home node stores the most recent version of the page, and provides this version to other nodes on request. In our system, the first node to access a page becomes the home node of the page. When a node r that is not the home node requires updated contents of a shared page p, it sends a **PAGE** request to p's home node. This **PAGE** request contains the vector time stamp indicating the version of the page required by r. The home node satisfies these requests by returning the appropriate copy of p to r.

Before letting r modify p, the protocol creates a *twin*, a copy of the original version of p received from the home node. When r issues a subsequent release, it sends the changes made to p back to the home node in the form of a *diff*, which is a run-length-encoding of the differences between the new version of p and its twin. The diff is sent in a **DIFF** message, which also contains r's logical clock. When the home node receives the **DIFF** message, it applies the changes to the version of p at the home node and updates entry r in its vector timestamp for p. Other nodes can then request a version of the page containing these updates using a **PAGE** request. The home node of page p may directly modify p without first creating a twin or generating a diff. When the home node issues a subsequent release, the home node's logical clock value is stored into the proper entry of p vector timestamp, indicating that a new version of the page has been created.

When a node q issues a release for a synchronization object l, the protocol creates a *write notice* for the pages modified by q since q issued its last release. The protocol guarantees that these write notices and previous ones created by q are sent to any node that subsequently performs an acquire for object l. When a node r receives a write notice for a page p, it updates its vector time stamp for p. If the version corresponding to the new timestamp is not already present at r, the protocol uses VMM protection mechanisms to prevent r from accessing p. If r attempts to do so, an access fault is generated, causing the protocol to fetch the latest version of p from the home node.

When the home node of a page p modifies the page and issues a subsequent release, the protocol sends write notices as described above and then places p into *exclusive state*. Pages in exclusive state remain writable on the home node and do not generate write notices at a release. A page remains in exclusive state until a remote node requests an updated copy of the page, which puts the page back into *shared state*. The exclusive state is an optimization that allows the home node to modify the page over several intervals without incurring the costs of multiple page faults or write notice messages. Correctness is preserved because a remote node that needs later updates must also see the earlier changes, and thus must request a new version of the page from the home node.

The protocol we describe here is a *multiple-writer protocol* because it allows multiple nodes to update the same page concurrently. Multiple writer protocols address the issue of *false sharing*, which occurs when two different

nodes simultaneously require write access to separate structures that happen to reside on the same page. If the program properly serializes writes to shared memory with synchronization operations, all concurrent writes will operate on different portions of the page. These writes will then be merged into a single version when the diffs are applied to the page at the home node.

3 A Speculative Protocol for Software DSM

We describe our speculative protocol by discussing how it addresses the three challenges faced by all techniques based on speculation, namely, 1) how to generate predictions, 2) how to act on predictions, and 3) what to do when a prediction is wrong.

3.1 How we generate predictions:

Predictions are generated by a memory sharing predictor which uses patterns in past shared memory accesses to guess future memory accesses. Our predictor, shown in Figure 1, uses a two-level structure similar to that used by Lai & Falsafi [12] for their memory sharing predictors for hardware DSMs. This two-level structure was inspired by the two-level adaptive-training branch prediction scheme of Yeh & Patt [21]. Our predictor maintains a history table and a pattern table for each shared page at the page's home node. Maintaining predictor state at the home node is a natural choice for a home-based protocol, since the home node processes all requests for a page.

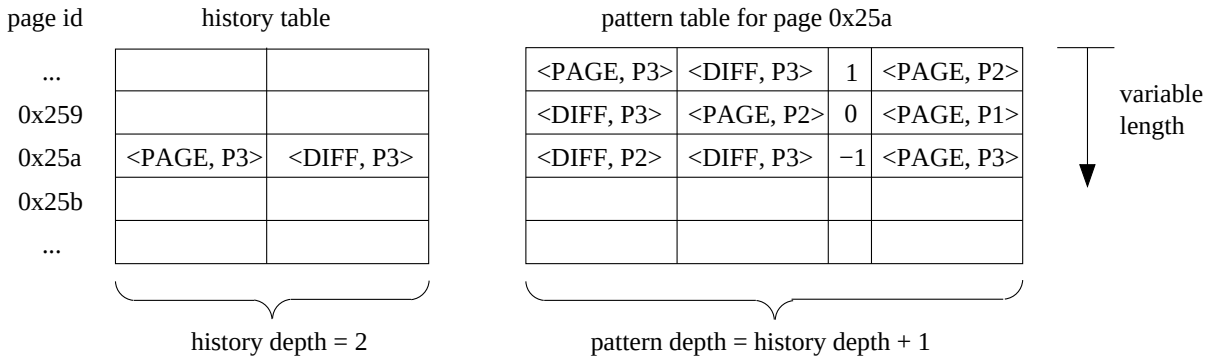

Figure 1. Two-level predictor for memory accesses

The *history table* is a record of the n most recent **PAGE** or **DIFF** requests processed for each page of shared memory, where n is the *history depth*. For each request, the history table records the request type and the node making the request. For example, the history table in Figure 1 indicates that the last two requests processed for page 0x25a were a **PAGE** request from node P3 followed by a **DIFF** request from node P3. Recall that **PAGE** messages specify the version of the page required by the requester. If the version of the page requested is not yet available

at the home node, the **PAGE** request must be deferred until the required **DIFF** messages have been received and processed. For this reason, **PAGE** requests are recorded in the history table at the time they are processed, rather than at the time they are received.

The *pattern table* is a record of all observed patterns of $n + 1$ **PAGE** or **DIFF** requests processed for a page. We maintain a separate pattern table for each shared page. This allows each page to have distinct sharing patterns at the expense of increased storage cost for pattern tables and longer training time (the time required to observe enough patterns to be able to predict future memory accesses). The pattern table has a variable length to support pages with many different request patterns. In Figure 1, the pattern table for page 0x25a contains three request patterns. The history and pattern tables can be used to predict the next **PAGE** or **DIFF** request by finding a pattern in the pattern table whose initial n requests match the sequence of requests in the history table for the page. If such a pattern is found, our protocol predicts the $n + 1$st request in the pattern as the next request for the page. Each pattern table entry also contains a two bit confidence indicator, shown in Figure 1 as a value of 0, 1, or -1. This confidence indicator is used by the feedback mechanism of our protocol, described in Section 3.3.

Special care must be taken to correctly record the accesses to a page performed by its home node. Since all updates are eagerly pushed to the home node of a page, no **PAGE** request is generated when the application on the home node attempts access to the page. Furthermore, the application can directly update the copy of the page at the home node, so no **DIFF** message is needed to make these updates available to other nodes. Still, it is important that these accesses be recorded in the history and pattern tables so that they can be used to predict future accesses and trigger speculative actions. Therefore, a page fault on a page at its home node is recorded as a **PAGE** request, and updates made at a page's home node are recorded as a **DIFF** by the home node.

Note that **PAGE** requests are the only actions which can be performed speculatively. Therefore, as an optimization, only patterns that end in a **PAGE** request are placed in the pattern table. This allows the pattern table to consume less space and also makes pattern table searching more efficient.

We also consider a modified version of our predictor that is capable of recording multiple **PAGE** requests in a single request entry by storing requesting node information in a bit string rather than as node ids. This not only conserves space in the history and pattern tables, it also masks the order of the **PAGE** requests in the entry, which reduces predictor training time for applications with high levels of read sharing. Lai & Falsafi use this technique in their hardware DSM predictors and found it to be highly effective at reducing predictor state and training time. In the remainder of the paper, we refer to this form of the predictor as the *vector predictor* as opposed to the original *standard predictor*.

3.2 How we act on predictions:

Once a history of accesses for a page has been built up in the pattern table, the home node can use this information to predict future accesses and speculatively issue protocol operations. Conceptually, the home node should attempt to issue speculative protocol operations whenever a new version of the page is available. This can occur when the home node processes a **DIFF** request from another node, or after the lock release or barrier call that ends an interval in which the page was modified by the application running on the home node.

For barrier calls, nodes arriving at the barrier early issue speculative protocol operations while waiting for the remaining nodes to arrive at the barrier. This reduces the effective overhead of speculative processing by overlapping it with the latency of the barrier synchronization. However, once all nodes reach the barrier, the barrier synchronization completes, and any remaining speculative processing is performed after all nodes are allowed to depart from the barrier. This ensures that speculative processing does not increase the latency of barrier synchronization.

To execute a speculative protocol action, the home node of the page sends the updated version of the page to the predicted node using a **SPEC** request message. In addition to the page data, the **SPEC** request contains the page address and its current vector time stamp. When speculative actions are performed, the home node records them in the history table as if they had been triggered by an actual **PAGE** request from the remote node. Failing to do so could lead our predictor to observe false patterns. In particular, if a **SPEC** request were not recorded in the history table but succeeded in avoiding a remote page miss, the predictor would record this as a pattern in which the remote node did not require the data, and thus could fail to predict the **PAGE** request in future iterations.

When the remote node receives a **SPEC** request, it first checks the timestamp supplied in the request to ensure the data can be accessed by the application on the remote node. If this check fails, it triggers the feedback mechanism described in Section 3.3. Otherwise, the local copy of the page is updated with the data supplied in the **SPEC** request. The manner in which the update is performed depends on the state of the page at the time the **SPEC** request is processed. If the page is not accessible to the application, the data is simply copied into the page and the page is made readable. If the page is currently readable to the application, but not writable, again the data is simply copied to the page. As long as the application properly serializes accesses to shared data (a basic assumption of release consistency), we know that it is not accessing any portions of the page containing updated data and thus will perceive no changes when the new data is copied into the page.

If the page is currently writeable by the application, special care must be taken in updating the local copy of the page to ensure that updates made in the local copy are not lost. In this case, we generate a diff between the twin of the page and the page contents supplied in the **SPEC** request. This diff is then applied to the local copy of the

page. We also update the twin with the page contents supplied in the **SPEC** request so that only the local node's updates are returned to the home node.

Finally, at the time the **SPEC** request arrives, the remote node may have already issued a **PAGE** request to obtain the version of the page provided in the **SPEC** request. In this case, the predictor correctly predicted the access, but did not predict it early enough to avoid a page miss by the application at the remote node. However, our protocol still avoids some portion of the remote access latency by updating the page with the data provided in the **SPEC** request and allowing the application to resume processing. When the response to the **PAGE** request arrives at the remote node, it is discarded. We refer to these cases as *partial page misses*, since a page miss is resolved without incurring a full remote access latency.

3.3 What we do when the prediction is wrong

Our speculative protocol includes a feedback mechanism to identify and suppress incorrect speculative actions. A speculative protocol action is incorrect if the remote node does not access the supplied data before it is invalidated by a subsequent write notice. To detect all incorrect speculative actions, we would have to detect any accesses to data supplied speculatively. This would require read-protecting the data so that an access generates a VMM protection exception. Read-protecting pages from speculative actions is not required by the memory consistency model, so correct speculative actions would incur unnecessary read-access page faults if this approach were used to detect incorrect speculative actions. To minimize the impact to correct speculative actions, we use a different approach that identifies some but not all incorrect speculative actions. Our approach focuses on one particular category of incorrect speculative actions — when the remote node has already received a write notice invalidating the data when the **SPEC** request arrives.

Incorrect speculative actions are detected by the remote node when it verifies the timestamp in a **SPEC** request against its local timestamp for the page. If the remote node determines that a **SPEC** request was incorrect, it sends a message back to the home node of the page to inform it of the incorrect speculation. When it receives a feedback message, the home node clears the confidence indicator field of the pattern table entry that triggered the incorrect action. The confidence indicator acts like a two-bit saturating counter commonly used in hardware predictors by inhibiting the pattern from triggering speculative operations until the pattern is re-established.

4 Implementation

We modified an existing home-based LRC DSM system, HLRC from Rutgers University [17], to use our new speculative home-based release consistency protocol, SHRC. In addition to modifying the basic DSM protocol, we also converted the system to use standard UDP interfaces for network communication instead of Virtual Interface

Architecture (VIA) networking. We chose to use UDP over VIA because UDP can be used with commodity network infrastructures (e.g. ethernet) whereas VIA requires special purpose network interface cards and switches. Since UDP does not support reliable communication, we added the necessary windowing and retransmission logic to protect against dropped packets. However, in our evaluation, we increase the socket buffer sizes and queue lengths to ensure no packet losses, so that performance results are meaningful and repeatable. To ensure **SPEC** requests do not delay processing of other, higher priority requests such as **PAGE** or **LOCK** requests, all **SPEC** messages are sent to a separate socket, and messages from this socket are only processed when there are no outstanding messages on the socket for the base protocol.

HLRC uses a page table to maintain the state of each shared page on a node. Rather than create a separate history table, we incorporate the history table entry for each page into its page table entry. We also add a pointer to the pattern table, which is stored as a list of fixed size segments. With this organization, the pattern tables for most pages can be small, but can grow to accommodate a large number of patterns for the few pages where this is required.

Early experiments with our implementation revealed that variations in request orderings could create cycles in the pattern table that would result in redundant **SPEC** requests. For example, one pattern might indicate that a **PAGE** request from node 2 follows a **PAGE** request from node 1, and another pattern may have these reversed. Longer cycles are also possible. Using a larger history depth reduces this problem somewhat, but some applications still experience this problem for a history depth of 5. We address this problem in our implementation with an additional field in the page table entry for each page. The `uptodate` field contains one bit for each node which indicates if that node has the version of the page currently available at the home node. The `uptodate` bit for node r is set for page p if the home node previous sent the current version of p to r, or if r had the previous version of p and supplied the most recent diff applied to p. Before sending a **SPEC** request for page p to r, the home node checks the `uptodate` field for p, and squashes the **SPEC** request if r already has the current version of p.

5 Performance Evaluation

5.1 Methodology

Our evaluation environment consists of a cluster of 8 machines, each having an 866 MHz Pentium III processor, 1GB of SDRAM memory, and a gigabit Ethernet adapter. The machines are connected using an Extreme Networks BlackDiamond gigabit Ethernet switch. All machines are running the RedHat 7.3 Linux OS with a 2.4.18 version kernel. The performance of certain basic operations of the system are presented in Table 1.

We evaluate SHRC using eight shared-memory benchmark applications. The applications and their relevant

remote access miss	370 usecs
local page fault	48 usecs
mprotect of 4K page	0.98 usecs
copy of 4K page	6.9 usecs

Table 1. Performance of basic operations

Application	Input Parameters
`barnes`	4096 bodies, 6 timesteps
`cholesky`	input file tk15.O
`em3d`	32000 nodes, 15% remote, 50 timesteps
`fft`	64 x 64 x 64 array, 16 iterations
`ocean`	130 x 130 array, 60 iterations
`radix`	8 million integers, max = 16777215, radix 256
`tomcatv`	128 x 128 array, 50 iterations
`water-sp`	512 molecules, 12 timesteps

Table 2. Applications and input parameters

input parameters are summarized in Table 2. We use five applications from the SPLASH-2 Benchmark suite [20] used in the evaluation of the HLRC DSM system [17]. `Barnes` simulates gravitational forces on a collection of bodies in three dimensions using the Barnes-Hut hierarchical N-body method. The bodies are assigned to processors according to their position in three dimensional space, which is represented using a hierarchical data structure called an octree. `Cholesky` performs blocked Cholesky Factorization on a sparse matrix. `Ocean` is the non-contiguous-partitions version of the SPLASH-2 ocean application. `Radix` performs the standard radix sort algorithm on an array of integers. `Water-sp` is a molecular dynamics application that simulates the motion of water molecules in three dimensional space.

Two applications come from the suite of benchmarks used by Lai & Falsafi in their work on speculation in hardware DSMs [12]. `Em3d` is a shared-memory implementation of the Split-C program to perform 3D modeling of electromagnetic waves [7], and `tomcatv` is a shared-memory implementation of the mesh generation program from the SPEC92 floating-point benchmark suite. The final application is `fft`, a three-dimensional FFT kernel from the NAS parallel benchmarks [3]. The version we use comes from the Treadmarks application suite [2].

Selected statistics from executions of these applications on the base HLRC DSM protocol are shown in Table 3.

Application	memory size (KB)	exec time (ms)	barriers	lock acqs	messages	traffic (KB)
barnes	4,971	1,385	14	2	3,666	4,684
cholesky	20,296	4,040	3	1,409,004	11,952	9,218
em3d	2,708	3,925	100	0	15,484	31,754
fft	3,082	1,807	31	0	3,460	6,896
ocean	5,883	3,791	1,593	295	11,058	14,207
radix	15,676	1,311	11	24	8,416	14,407
tomcatv	347	382	196	0	342	679
water-sp	546	5,293	63	59	13,707	5,141

Table 3. Application characteristics

For all applications, statistics exclude initialization processing. In addition, statistics for iterative applications exclude the first iteration. This is the typical approach used for the SPLASH2 benchmarks, where the first iteration is excluded to eliminate startup effects. In practice, most iterative DSM applications are run for many iterations, and thus the performance of initial iterations have only a marginal impact on overall run time.

5.2 Performance Results

Figure 2 presents the performance results for our eight benchmark applications. The figure shows application execution times normalized to the execution time without speculation, indicated by the bar labeled nospec. The figure presents results for the standard predictor with history depths of 3 and 4, labeled spec_hd3 and spec_hd4 respectively, and the vector predictor with history depths of 2 and 3, labeled vmsp_hd2 and vmsp_hd3 respectively. We also ran experiments for other configurations, but the ones presented achieve the best performance improvements. In Section 5.4 we analyze a broader range of configurations to determine how predictor configuration affects application performance.

All results in this section are the average of five program executions. Figure 2 indicates the 95% confidence interval for the actual improvement, determined using a paired t-test for unequal means. Using separate experiments, we determined that statistical confidence was not significantly improved by using ten program executions for each configuration, indicating that these confidence intervals reflect the inherent variability in execution times for our benchmark applications. For four of our eight applications, SHRC achieves a statistically significant performance improvement in all four configurations shown. The vector predictor configurations also achieve statistically significant performance improvement for barnes and water-sp. On the other hand, performance is essentially

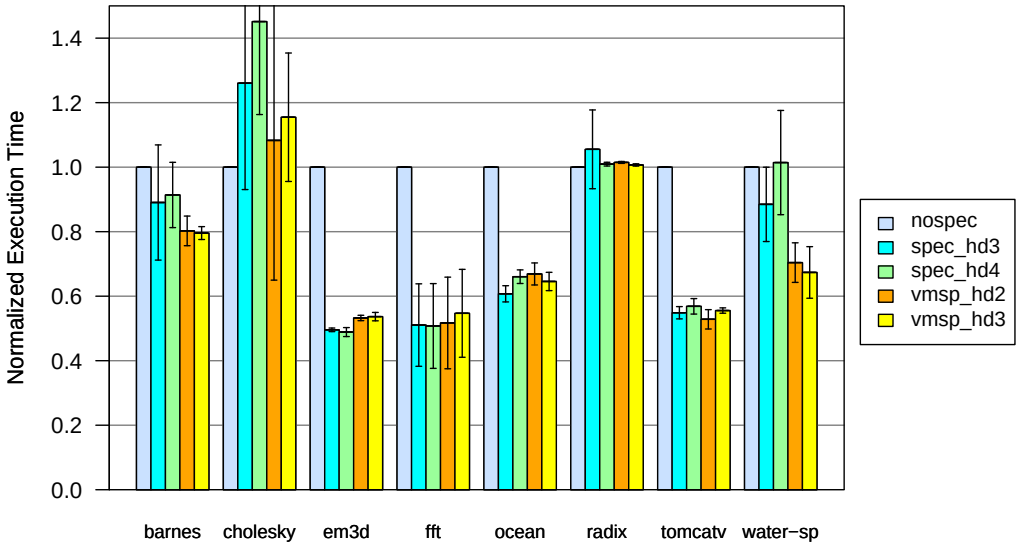

Figure 2. Performance benefits from SHRC

unchanged for `radix` and degrades significantly for `cholesky`.

For three applications, `em3d`, `fft`, and `tomcatv`, execution times are reduced by 40% to 50%. These applications exhibit very regular access patterns independent of the input data values, and for these cases SHRC works very well. In `em3d`, the problem domain is statically partitioned across processors and sharing is coarse grained. Furthermore, `em3d` is a single-writer application, meaning that all write accesses to a data item are performed by a single processor. Our protocol selects the first node to access a page as its home node, which in `em3d` is always the only node that writes to the page, and thus `em3d` exhibits no write sharing in our system. `Fft` uses barrier synchronization exclusively, and therefore benefits from our approach to hiding the overhead of speculative processing within the barrier protocol. `Tomcatv` is a stencil application in which processors read data produced by their nearest neighbors in a very regular pattern.

Speculation reduces execution times for `ocean` by 30% to 40%. `Ocean` is a single-writer application with coarse grain sharing. It makes heavy use of barrier synchronization and therefore also benefits from the integration of speculative actions with barrier processing.

Speculation using the vector predictor configurations also improves the performance of `barnes` and `water-sp` by 20% to 33%. Average execution time for the standard predictor configurations also decrease, but these results were not statistically significant at the 95% confidence level. Our executions of `barnes` perform only six iterations, and data access patterns may vary over the course of an execution since computation is divided across

the nodes according to the position of the bodies in the octree, which may change in each iteration. In `water-sp`, the set of water molecules is partitioned spatially, and each processor is assigned a spatially contiguous region to process. Sharing occurs when processors must compute the effect of molecules in neighboring regions on molecules within their assigned region. Molecules may also move across regions, but this occurs much less frequently. Therefore, access patterns in `water-sp` are generally quite regular, with one processor, typically the home for the page, making updates and all adjacent processors in the grid then reading the updated data.

SHRC is ineffective for improving the performance of `radix`. This is largely because our executions of `radix` make only three passes over the input data, one for each "digit" (base 256) in the values to be sorted. This means that there is little opportunity to develop a set of patterns that are useful for speculative protocol actions. Furthermore, the reference patterns in `radix` are quite irregular since they depend on the values to be sorted.

Finally, execution times actually increase for `cholesky`. This program uses a task queue to distribute work to processors, resulting in highly irregular access patterns. Furthermore, synchronization for the task queue is handled using locks, leading to extremely high lock activity (an average of 500,000 lock acquires per second in our executions). This combination of factors makes SHRC unsuitable for this application.

For the three applications with the largest performance gains, the vector predictor configurations do not improve performance over the standard predictors; performance actually degrades slightly for `em3d`. These applications do not exhibit high degrees of read-sharing, and therefore do not benefit from the vector predictor. Compared to the standard predictor, the vector predictor achieves significantly improved performance for `barnes` and `water-sp` because of higher levels of read-sharing in these applications.

Overall, the vector predictor delivers more consistent performance improvement than the standard predictor. The vector predictor resulted in statistically significant performance improvement for six of our eight applications, compared to only four for the standard predictor. Furthermore, while the standard predictor does marginally outperform the vector predictor for some applications, the vector predictor substantially outperforms the standard predictor for several others. On this basis, we conclude that the vector predictor achieves better overall performance improvements for our benchmark applications.

5.3 Protocol Efficiency and Effectiveness

The primary metric of efficiency of our speculative protocol is accuracy, which is the percentage of speculative actions that are correct. We measure accuracy using two different approaches. First we measure the accuracy of the predictor in isolation, that is, without any speculative actions performed by the protocol. We do this by maintaining the predictor history and pattern tables, but without performing any speculative requests, and comparing the predicted **PAGE** requests to the actual **PAGE** requests for each page. In this approach, we compute

accuracy as the ratio of correct to all page miss predictions. In the second approach, we measure accuracy for the speculative protocol. Here we estimate the number of correct speculative actions by the reduction in page misses relative to an execution without speculation, and then compute accuracy as the ratio of correct speculative actions to all speculative actions.

Figure 3 presents the accuracy achieved by the predictor in isolation. The predictor achieves an accuracy of 90% or higher for five of our eight applications: `em3d`, `fft`, `ocean`, `tomcatv`, and `water-sp`. The predictor works well for `em3d` and `fft` because both exhibit producer/consumer style sharing, where pages are seldom read-shared by multiple nodes. Coarse-grain sharing also improves accuracy, as illustrated in the results for `ocean`, by minimizing false-sharing effects. Prediction accuracy for `tomcatv` is high in part because the application carefully aligns and pads its data to minimize false sharing. Prediction accuracy for `water-sp` is also high, because of very regular access patterns to the water molecule structures. However, in each iteration `water-sp` performs several reductions whose results are accumulated into globally shared variables. Locks are used to serialize access to these global variables, and the order of access can change depending on the order in which the nodes arrive at the acquire for the lock. Therefore, access patterns for these global variables could be somewhat irregular and may account for the incorrect predictions experienced by `water-sp`.

All predictor configurations achieve accuracy of 55% to 70% for `barnes`. A significant portion of the incorrect speculative actions for `barnes` are caused by the *last speculation effect*, which refers to the speculative actions performed in the final phase of the application. Since the application is about to complete, these actions are unnecessary and are counted as incorrect speculative actions. This effect reduces predictor accuracy for `barnes` by 18% to 36% for the configurations shown. Finally, we see that prediction accuracy is poor for `cholesky` for all predictor configurations, and varies considerably for `radix`. For `cholesky`, this is caused by the inherent irregularity of access patterns, which are effectively randomized by the task queue mechanism used to distribute work to processors. As noted above, access patterns in `radix` are highly data dependent, making them difficult to predict.

Figure 4 presents the accuracy achieved by the speculative protocol. The hatched portion of each bar indicates the portion of correct speculative actions that resulted in partial page misses. In comparison with Figure 3, we see in Figure 4 that accuracy of the speculative protocol is considerably lower for `ocean`, `tomcatv`, and `water-sp`. We also note that `ocean` and `tomcatv` experience the highest occurrence of partial page faults. This suggests that prediction accuracy is low for these applications because the protocol cannot execute speculative actions quickly enough to avoid the remaining page misses.

Partial page misses are quite low for `water-sp`, indicating that there is another factor leading to reduced ac-

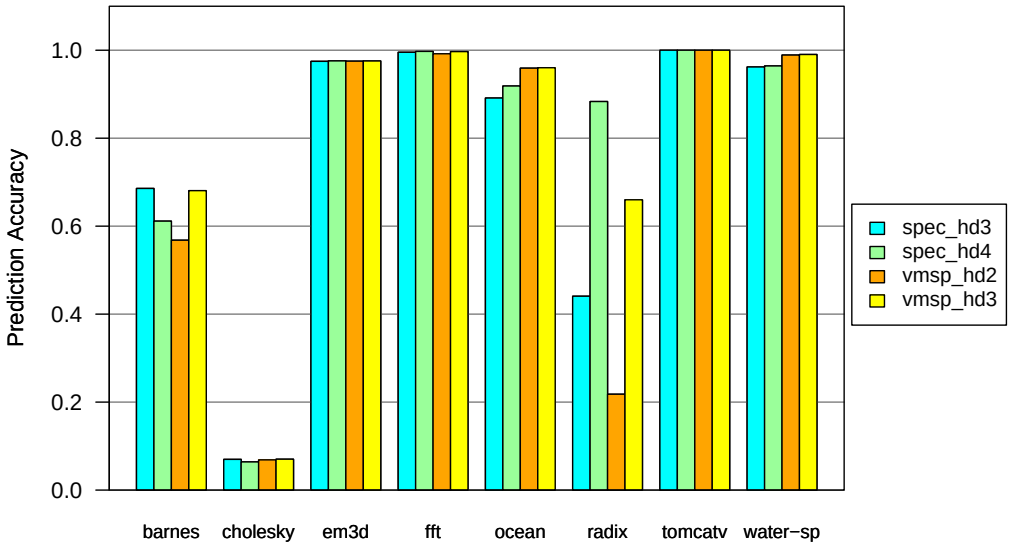

Figure 3. Accuracy of the predictor in isolation

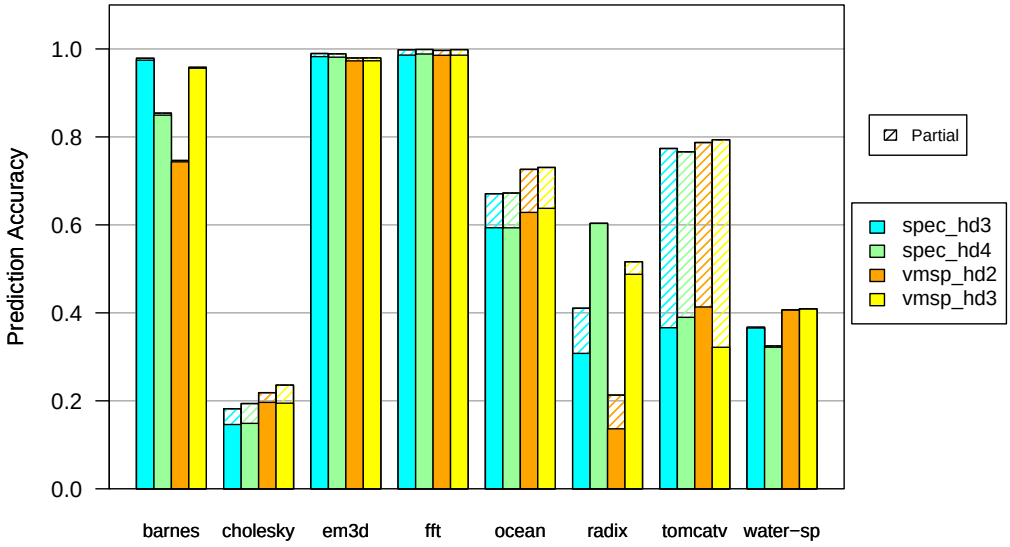

Figure 4. Accuracy of the speculative protocol

curacy for the speculative protocol. Detailed instrumentation revealed that `water-sp` experiences a significantly higher rate of write faults when speculation is enabled. This clue helps expose the cause of the reduced accuracy for the speculative protocol, which is that pages in exclusive state are being transitioned to the shared state when speculative actions are performed. This is necessary to ensure correctness, but defeats the optimization of placing pages in exclusive state, resulting in unnecessary write faults and write notices. This leads to reduced accuracy because a new version of the page, and thus new predictions, are generated for each interval in which the page was modified. In the case of `water-sp`, molecules are updated by the home node over a series of three intervals before the updates are actually used by remote nodes. As a result, the speculative protocol generates three times more **SPEC** requests than required, effectively reducing accuracy by 66%.

In most applications, our feedback mechanism detected very few incorrect speculative operations. However, between 6% and 10% of the speculative operations for `cholesky` were determined to be incorrect. This explains why the speculative protocol achieves higher accuracy for `cholesky` than the predictor in isolation. While this application was clearly the worst performing of those we study, this indicates that performance could well have been worse without our feedback mechanism. Up to 10% of speculative operations for `radix` are also found to be incorrect, but this does not lead to improved predictor accuracy, probably because of extremely low page miss coverage, as explained below. In the remaining applications, less that 0.1% of speculative operations were incorrect for all configurations except the standard predictor configurations of `ocean`, in which approximately 1% of speculative operations were incorrect.

An important measure of the effectiveness of our speculative protocol is page miss coverage, which is the percentage of page misses that are eliminated by speculation. We measure page miss coverage achieved by the predictor in isolation and by the speculative protocol as a whole. Figure 5 presents the miss coverage achieved by four configurations of the predictor, with speculative actions disabled, for our eight benchmarks. The page miss coverage for `em3d`, `fft`, `ocean`, and `tomcatv` exceeds 90% for all configurations. These four applications also experienced the highest performance gains from speculation. Likewise, the two worst performing applications, `cholesky` and `radix`, have the lowest page miss coverage. This demonstrates the importance of page miss coverage in achieving performance gains from speculation.

Figure 6 presents the page miss coverage of the speculative protocol. Page miss coverage is computed as the ratio of correct speculative actions to the number of page misses without speculation, where the number of correct speculative actions is estimated by the reduction in page misses relative to an execution without speculation. The hatched portion of each bar again represents correct speculative actions that resulted in partial page misses. Results are generally similar to those achieved by the predictor in isolation.

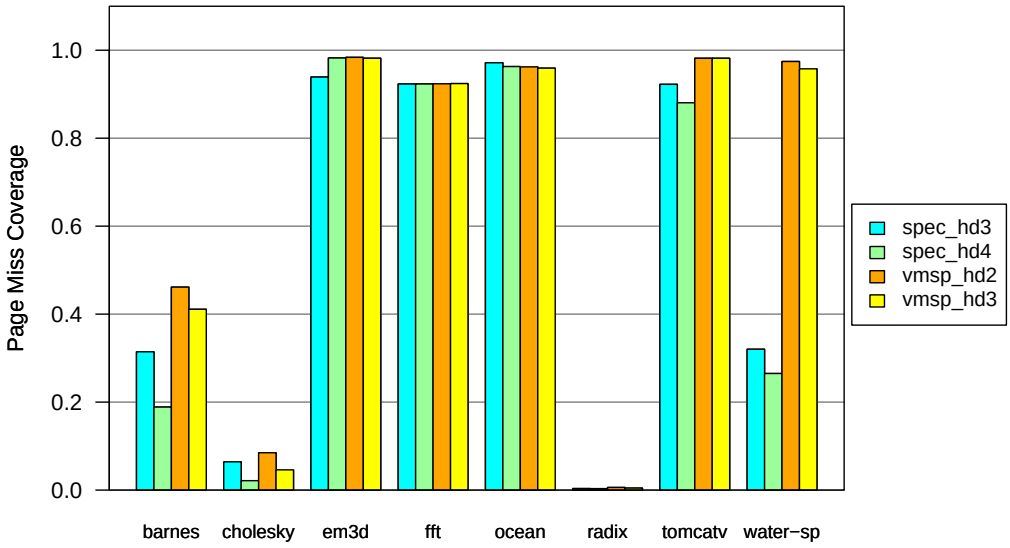

Figure 5. Page miss coverage of the predictor in isolation

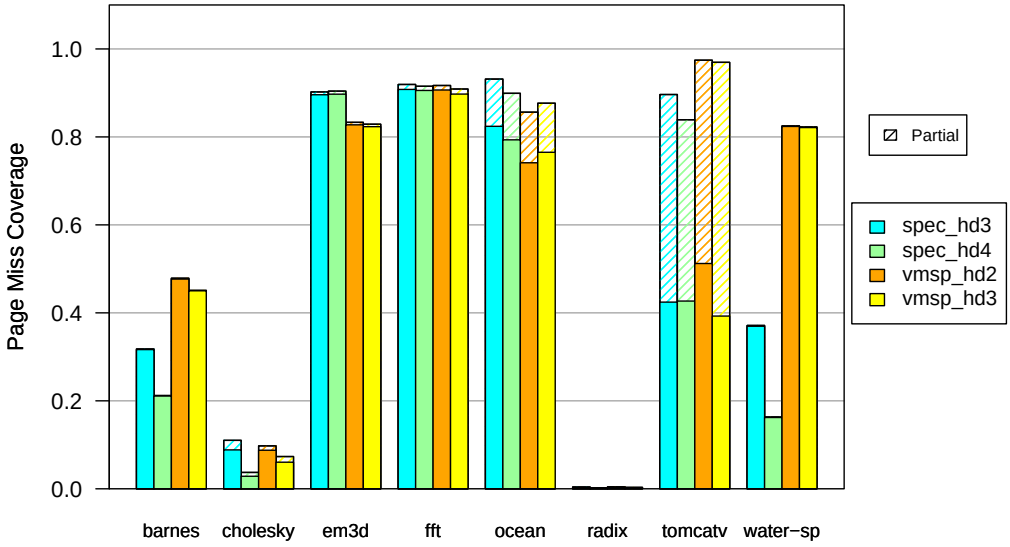

Figure 6. Page miss coverage of the speculative protocol

5.4 Effect of History Depth

Next we consider the effect of the history depth parameter on the operation of our speculative protocol. Recall that history depth determines the number of access history entries used in predicting a subsequent access for a page. Figure 7 illustrates the effect of increasing standard predictor history depth on application performance, prediction accuracy, and miss coverage for four of our benchmark applications. In general, as history depth increases, we expect predictor accuracy to improve since predictions are based on more information. At the same time, coverage decreases since the predictor generates more patterns, which increases the training time. Performance could therefore improve or degrade with increasing history depth, depending on whether increased accuracy or reduced coverage is the dominant factor. `Barnes` is a good illustration of these general trends, except that accuracy drops for history depth greater than three. In applications with irregular access patterns, such as `barnes`, increasing history depth may eliminate more correct speculative operations than incorrect ones, leading to reduced accuracy. The results for `fft` also follow these general trends, but with a more abrupt rise in accuracy between history depth of 2 and 3. This indicates that several frequently occurring access patterns share a common sequence of two accesses, but no common sequences of three accesses, which allows the predictor with a history depth of 3 to disambiguate these access patterns.

`Em3d` illustrates another common trend. This application is relatively insensitive to history depth. This is because the access patterns of `em3d` are so regular that nearly perfect accuracy can be achieved with a history depth of one. Furthermore, because of these regular access patterns, the pattern space stays relatively small, limiting the loss in coverage. `Water-sp` shows a more unusual trend with increasing history depth, with both accuracy and coverage decreasing with increasing history depth. As a result, performance for this application is best for small history depths.

Figure 8 illustrates the effect of history depth for the vector predictor. Here results are much more consistent, with all four applications performing at or near their best levels for history depth of 2. For `barnes`, accuracy improves for larger history depths, but its effect is offset by reduced1 coverage. For `em3d` and `fft`, the vector predictor achieves nearly perfect accuracy at a history depth of 2, so these applications benefit little from larger history depths. Accuracy is relatively low for `water` and appears insensitive to history depth, and thus performance is better for small history depths where coverage is greater.

The conclusion from this analysis is that both the standard predictor and vector predictor can achieve sufficient accuracy and coverage at low history depths. Over all the applications in our study, the optimal history depth for the standard predictor appears to be either 3 or 4, and the optimal history depth for the vector predictor is either 2 or 3.

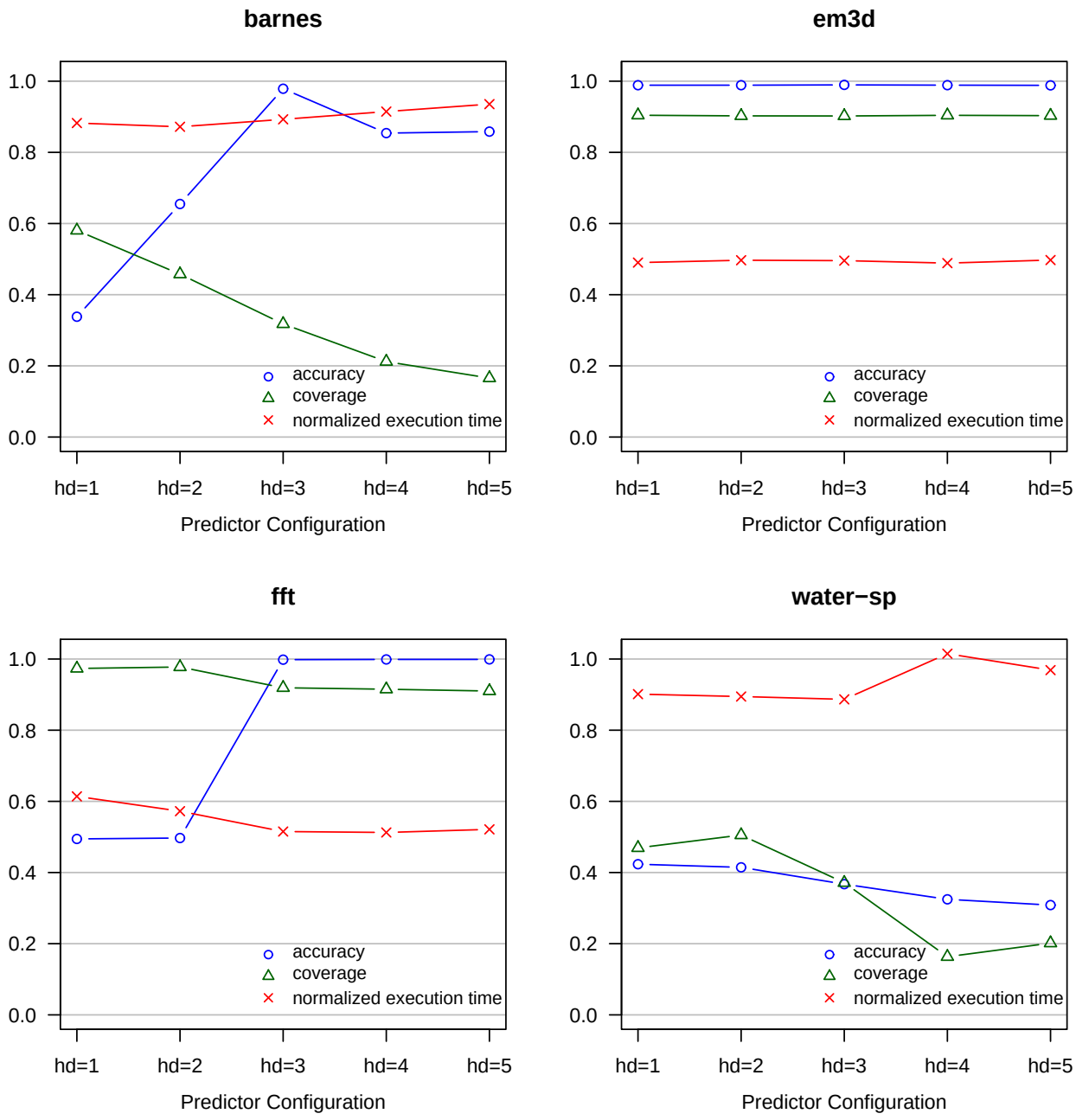

Figure 7. Effect of standard predictor history depth

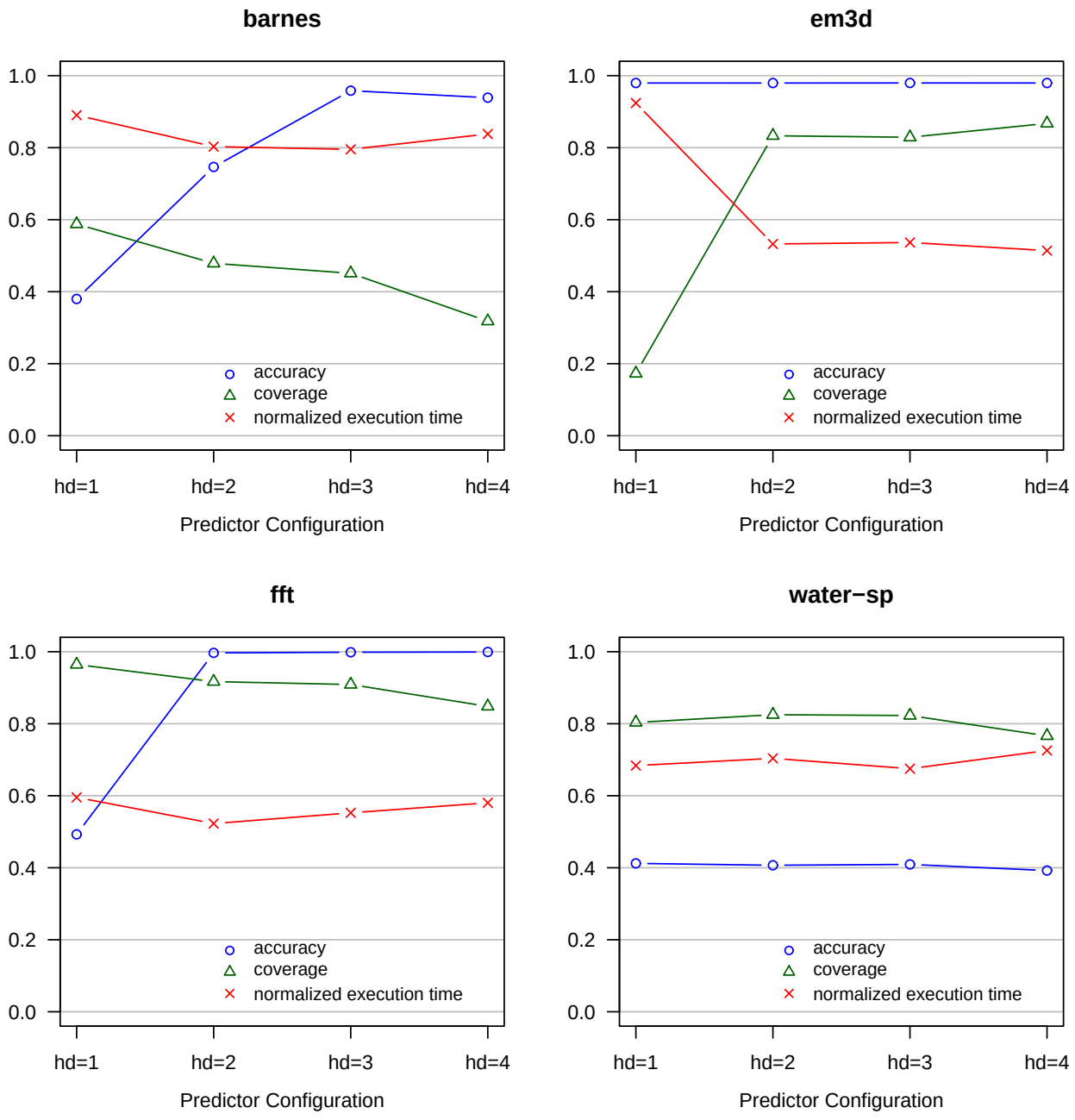

Figure 8. Effect of vector predictor history depth

5.5 Protocol Overhead

Here we briefly report the overheads of our speculative protocol. Figure 9 presents the number of messages sent by the DSM for each configuration, normalized to the message count of an execution without speculation, and broken down into **DIFF** messages, **PAGE** messages, and **SPEC** messages. Note that the number of **DIFF** messages is identical for all configurations, indicating that SHRC does not generate any additional **DIFF** messages. Some applications, most notably `tomcatv` and `water-sp`, do experience increased message counts for the speculative protocols. However, speculation results in greatly reduced message counts for some applications, such as `em3d` and `fft`. This is because **PAGE** requests always require a **PAGE** response message, whereas correct **SPEC** requests require only an occasional **SPEC** response for flow control. This reduction in message count is partly responsible for the significant performance improvement provided by speculation for these two applications. Even applications that experience large increases in message counts can benefit from speculation, as illustrated by `water-sp`, for which speculation using the vector predictor can reduce execution time by 30%.

Figure 10 presents the aggregate size of the protocol messages sent by the DSM for each configuration, normalized to aggregate message size of an execution without speculation, and broken down into **DIFF** messages, **PAGE** messages, and **SPEC** messages. Again we note that **DIFF** traffic is not increased by our speculative protocol. Aggregate message size is virtually unchanged for `em3d` and `fft`, because of the high accuracy achieved by the predictor for these applications. Note however that 80% to 90% of **PAGE** traffic has been replaced by **SPEC** traffic. Aggregate message size for `radix` is also virtually unchanged, but this is primarily because of extremely low coverage. At the other extreme are `cholesky`, `ocean`, `tomcatv` and `water-sp`, which send much more data in speculative configurations than the non-speculative case. The increase in aggregate message size consists of incorrect **SPEC** requests or **SPEC** requests for partial page misses, which do not eliminate the corresponding **PAGE** messages. This further illustrates the importance of predictor accuracy, since the overhead of speculation increases considerably as predictor accuracy decreases. Note that `tomcatv` achieves 40%–50% performance improvement despite a 60% increase in aggregate message size. This indicates that application performance is not constrained by network bandwidth in our environment, and that speculation can be used to trade available network bandwidth for reduced network latency, thereby improving performance.

Speculation also requires additional storage to maintain history and pattern tables and additional protocol state. In our implementation, the page table is used to store the history table and per-page protocol state. This adds $n+4$ words to each page table entry, where n is the history depth. In a 32 node system for $n = 3$, this amounts to less than a 1% increase in storage for the history table and protocol state.

The amount of storage used by the pattern table depends on the predictor type (standard or vector), the history

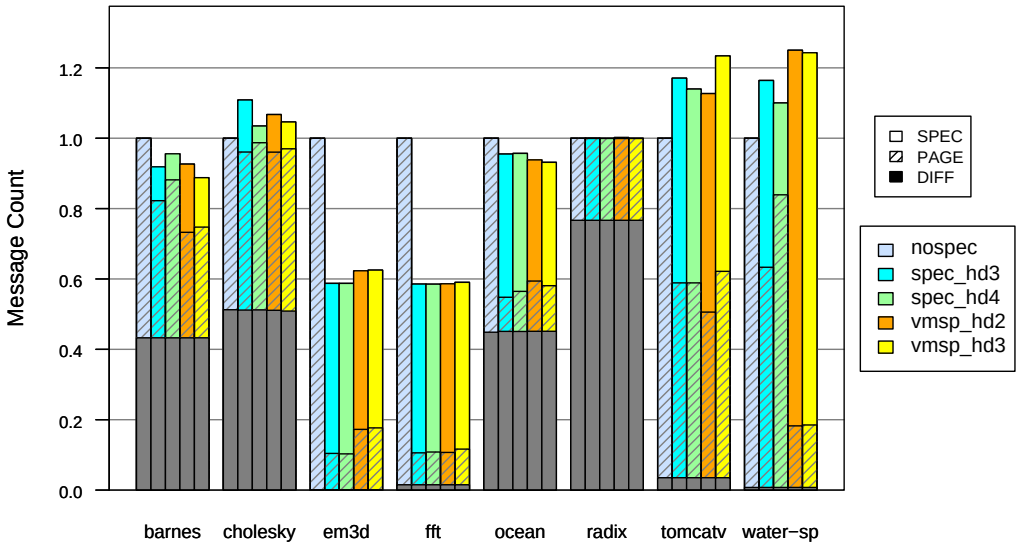

Figure 9. Message count

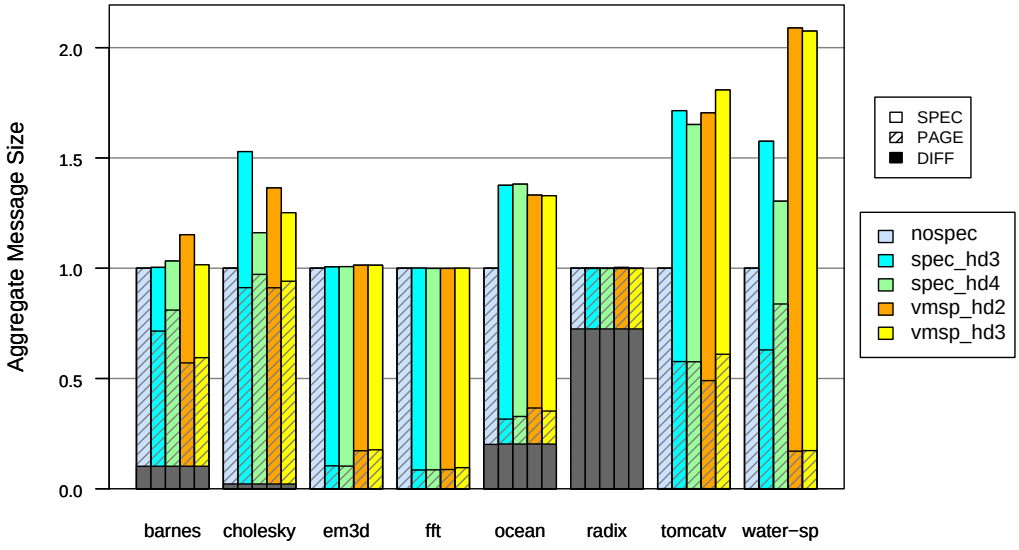

Figure 10. Aggregate message size

depth, and the application access patterns. The pattern table is stored as a list of fixed size segments, where segments are allocated only as needed. In applications with irregular access patterns or high levels of read sharing, the standard predictor can generate very large pattern tables. For example, for `water-sp`, the pattern tables for the standard predictor with history depth of 3 increase storage consumption by almost 6%. For the remaining applications in our study, standard predictor pattern table storage is always less than 1% of additional storage. The vector predictor encodes multiple **PAGE** requests in a single entry, thereby consuming less space. Storage consumption for the vector predictor is consistently less than that for the standard predictor, requiring at most 0.6% of additional storage (again for `water-sp` with history depth of 3). These storage requirements appear quite acceptable given the performance gains that can be achieved from speculation.

5.6 Summary of Performance Results

We find that both the standard and vector predictors achieve statistically significant performance improvement for several of our benchmark applications, have high efficiency and effectiveness for iterative applications with reasonably regular access patterns, and have low storage overhead. The vector predictor achieves more consistent performance improvement for our applications, and has substantially lower storage overhead, indicating it should provide superior results for most applications. Further performance improvement may be possible by reducing the negative impacts of the last speculation effect, the loss of the exclusive mode optimization, and partial page misses.

6 Related Work

One area of related work is the use of speculation in the context of hardware DSM systems. Lai & Falsafi developed *Memory Sharing Predictors (MSPs)*, a technique for predicting future memory accesses based on patterns of recent application/system memory reference behavior [12]. Simulation studies of MSPs show that they can achieve high prediction accuracies and reduce execution times for a suite of shared memory benchmark programs by 12% over the base consistency protocol. The predictor used in SHRC was inspired by MSPs but is designed for software DSMs implementing a release consistency memory model, which requires new approaches for dealing with multiple writers, explicit release semantics, large sharing units, and much larger latencies for remote memory accesses. Other related work in hardware DSM systems includes producer-initiated updates [1] and compiler generated prefetching [16].

A number of prefetching techniques have been proposed for LRC software DSM systems [4, 9]. The most recent is Delphi [18], a home-based LRC DSM which speculatively prefetches pages based on a history of previously accessed pages. Whenever a node must request updated data from another node, the access history is used to

predict up to N other pages that might be needed from the target node, where N is a fixed parameter of the protocol. A study of four applications showed that Delphi could improve performance by up to 14% over the base protocol without speculation. Our work is similar to Delphi in a number of respects. We implement speculation in the context of a home-based DSM, and we also employ a pattern-based predictor inspired by hardware-based mechanisms. However, our speculative protocol predicts which nodes will request a new version of a page, and then speculatively sends the page to these nodes. The key advantage of our approach is that speculative actions are triggered when a new version of a page is available, which helps to avoid speculative actions that are performed before the required version of the page is available.

The Lazy Hybrid (LH) protocol of Keleher in Treadmarks is a form of speculation that has been implemented within a software DSM system [11]. This protocol uses a history of past accesses to decide whether to update or invalidate shared data at remote nodes at the time of an acquire. This is a form of speculation since sending updates to a node is only beneficial if the node actually accesses the data before subsequent updates occur on another node.

Keleher proposed the the Lazy Hybrid (LH) protocol for the Treadmarks DSM system [11]. On an acquire, this protocol speculatively sends new versions of page contents to any node that previously accessed the page; write notices are sent to all other nodes to invalidate the page contents. Our protocol is different because we perform eager update (at the time of a release), and we base our predictions on patterns of access requests rather than just a single prior access by the remote node. Keleher has also proposed a barrier-only speculative protocol for applications with extremely regular access patterns [10]. Our protocol supports a wider range of synchronization mechanisms and more general access patterns.

Finally, the HLRC on VIA system [17] also has as its goal reducing latency of remote accesses, but by using improved communication mechanisms. The Virtual Interface Architecture (VIA) is specifically designed to reduce message latency by giving the application direct access to the network interface without context-switches into the kernel. VIA also provides support for zero-copy messaging and RDMA, further reducing send and receive overhead on the critical path. While this approach can considerably reduce the latency of remote misses, it does not eliminate access misses as our approach does, and it requires special hardware support (VIA-compliant network interface cards and switches), which are not necessary in our approach.

7 Conclusions

We present speculative home-based release consistency (SHRC), a speculative protocol for release consistent software DSM systems that seeks to improve application performance by reducing the latency of remote accesses. Our protocol employs a pattern-based predictor to determine what protocol actions to perform speculatively, uses

synchronization operations inherent to the RC memory model to trigger these speculative actions, and uses feedback to identify and avoid patterns that lead to incorrect speculative actions. Our performance evaluation using a suite of eight shared-memory benchmark applications demonstrates that performance improvements in the range of 40% to 50% can be achieved for applications with regular data access patterns. Applications with less regular access patterns can still achieve significant performance improvements of up to 30%. However, some applications receive no benefit from speculation and may even experience performance degradations. From these results, we conclude that software DSM systems should incorporate speculation to allow its use for those applications that can achieve significant benefits.

Acknowledgments

We would like to thank Liviu Iftode and Murali Rangarajan for making available their HLRC DSM system and the HLRC application suite. We also thank Babak Falsafi and An-Chow Lai for sharing their application suite for evaluating Memory Sharing Predictors for hardware DSMs. We also thank Ram Rajamony for many helpful discussions on the Treadmarks DSM system and software DSMs in general. Equipment for the performance evaluation was provided by IBM.

References

[1] H. Abdel-Shafi , J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fi ne-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[4] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software DSMs. In *International Conference on Supercomputing*, pages 385–392, 1998.

[5] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[6] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21th Annual International Symposium on Computer Architecture (ISCA-21)*, pages 106–117, April 1994.

[7] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993.

[8] L. Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, 1998.

[9] M. Karlsson and P. Stenström. Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared-memory systems. *Journal of Parallel and Distributed Computing*, 43(2):79–93, 1997.

[10] P. Keleher. Update protocols and iterative scientifi c applications. In *The 12th International Parallel Processing Symposium (IPPS)*, March 1998.

[11] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, October 1995.

[12] A.-C. Lai and B. Falsafi . Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA 26)*, pages 172–183, June 1999.

[13] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communication of the ACM*, 21(7):558–565, July 1977.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[15] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of SuperComputing '95*, December 1995.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, October 1992.

[17] M. Rangarajan and L. Iftode. Software distributed shared memory over virtual interface architecture: Implementation and performance. In *Proceedings of 4th Annual Linux Conference*, pages 341–352, October 2000.

[18] E. Speight and M. Burtscher. Delphi: Prediction-based page prefetching to improve the performane of shared virtual memory systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[19] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, pages 281–290, 1997.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[21] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.