Improving the Performance of Software Distributed Shared Memory with Speculation

Michael Kistler, *Member*, *IEEE*, and Lorenzo Alvisi, *Member*, *IEEE*

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 (SHRC), 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 release consistent software distributed shared memory system that runs on commodity hardware. We evaluate our protocol implementation using eight software distributed shared memory benchmarks and show that it can result in significant performance improvements.

Index Terms—Distributed shared memory, protocol design and analysis, speculation.

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 shared memory, the DSM system intercepts accesses to data that physically resides in a remote node and brings the data to the local node for processing. To improve performance, DSM systems commonly cache data from remote nodes in local memory, and implement a *coherence protocol* that ensures all copies of the data remain coherent even though they may be accessed concurrently across multiple nodes of the system. A *memory consistency model* specifies what values may be returned from the memory system based on prior memory reads and writes. The memory consistency model dictates the actions that must be performed by the coherence protocol. The most well-known memory consistency model 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 [1].

A key inhibitor to the performance of DSM systems is the latency of memory accesses that require coherence operations. 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], [3] 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.

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 commodity hardware. Unfortunately, software DSM systems often fail to provide performance comparable to either hardware DSM systems or to message-passing systems [4], [5]. This is primarily caused by the higher overhead of coherence operations in software DSM systems.

In this paper, we explore whether speculation can be used to narrow this performance gap by hiding the overhead of coherence operations by overlapping it with application execution. *Speculation* is the execution of an operation before it has been determined to be necessary or useful. 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 typically takes one of two forms: 1) predict the outcome of subordinate processing when it is required by the critical path 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 [6], value prediction [7], and data prefetching [8], [9].

The focus of our work is to study how speculation can be used to improve the performance of a software DSM system. In particular, we use patterns in past accesses to shared memory to speculate on which remote data will be required by the application. We then perform the protocol

• *M. Kistler is with the IBM Austin Research Laboratory, IBM Bldg. 904, 6E-000, 11501 Burnet Rd., Austin, TX 78758.*
E-mail: mkistler@us.ibm.com.
• *L. Alvisi is with the Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 78712. E-mail: lorenzo@cs.utexas.edu.*

Manuscript received 30 Aug. 2003; revised 30 June 2004; accepted 6 Nov. 2004; published online 21 July 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0148-0803.

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.

We make two main contributions. First, we present Speculative Home-based Release Consistency (SHRC), a new DSM protocol which speculatively updates data at remote nodes based on predictions of future memory accesses. Second, we describe an implementation of our protocol and report on its performance for a suite of eight benchmark DSM applications. We find that speculation can improve the performance of applications with regular access patterns by a factor of 1.6 to 2.0. For applications with less regular access patterns, our speculative protocol can still improve performance by a factor of 1.3. 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 release-consistent software DSM protocols, and Section 3 describes our SHRC protocol. Section 4 presents performance results for a prototype implementation of SHRC on a suite of benchmark programs. Section 5 summarizes the related work and Section 6 concludes the paper.

2 Background

Our speculative protocol is based on the Home-based Lazy Release Consistency (HLRC) protocol and uses a virtual memory management (VMM) page as the unit of sharing. A *home-based* protocol assigns each page of shared memory a *home node* which is responsible for maintaining and distributing data stored on the page to other nodes of the system. As its name implies, HLRC implements the release consistency memory model, and it is a lazy protocol because it defers sending updated data until it has been explicitly requested by another node. We extend this protocol by speculatively performing certain coherence operations before they are known to be required. In this section, we describe the design of release consistent DSM protocols and specific features of the home-based LRC protocol that are important for our purposes—a comprehensive discussion can be found in Iftode's dissertation [3].

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 are classified in the obvious manner. A barrier is classified as a release followed by an acquire. Each node maintains a *logical clock* [10] 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 ith entry of r's vector timestamp for p, then r should access a version of p that reflects all writes 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.

At the beginning of each interval, the protocol uses VMM protection mechanisms to prevent application writes to all pages of shared memory. When the application attempts to write to a shared memory page, the protocol is notified and takes appropriate action to record the access and make the page writable by the application. Before letting r write to 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** request, which also contains r's logical clock. When the home node receives the **DIFF** request, 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 changes using a **PAGE** request.

The home node of page p may directly modify p without first creating a twin or generating a diff. This is safe because proper synchronization ensures other nodes will not attempt to access the modified data until they have properly synchronized with the home node, and it is more efficient than modifying a private version and then creating and applying a diff to make the changes visible to other nodes. When the home node issues a subsequent release, the home node's logical clock value is stored into the proper entry of p's 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 ones previously created by q or previously received by q from other nodes 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

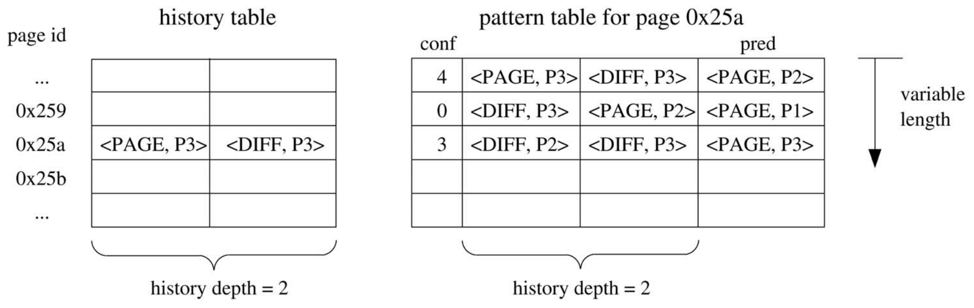
history table
page id
...
0x259
0x25a
0x25b
...
<PAGE, P3>
<DIFF, P3>
history depth = 2
pattern table for page 0x25a
conf
pred
4 <PAGE, P3> <DIFF, P3> <PAGE, P2>
0 <DIFF, P3> <PAGE, P2> <PAGE, P1>
3 <DIFF, P2> <DIFF, P3> <PAGE, P3>
variable length
history depth = 2

Fig. 1. Two-level predictor for memory accesses.

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 write to 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, concurrent writes on different nodes 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 present our speculative protocol by describing how it generates predictions and how it acts on these predictions.

3.1 How SHRC Generates 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 Fig. 1, uses a two-level structure similar to that used by Lai and Falsafi [11] for their memory sharing predictors (MSPs) for hardware DSMs. The predictor used in SHRC was inspired by MSPs, but is designed for use in a software LRC DSM. This requires new approaches for dealing with much larger latencies for remote memory accesses, large sharing units, a release consistent memory model, and support for multiple writers. Large remote access latencies offer greater opportunity for performance improvement, but also require new strategies for earlier prediction and partial latency avoidance. This is an important distinction between our work and MSPs, which do not allow speculatively sent data to satisfy a memory access that was not predicted early enough to avoid an access miss. Large coherence units in software DSMs result in lower prediction accuracy because of false sharing, but also allow a larger access history to be maintained per shared block, which could improve prediction accuracy. The relaxed consistency model used in most software DSM systems requires applications to use special synchronization operations that can provide the prediction mechanism with important clues on the timing of speculative operations. Finally, supporting multiple writers is a challenge because it requires an approach to merge speculative updates into a block that the target node may be in the process of updating.

Our predictor maintains a history table and a pattern table for each shared page at the page's home node. A home-based protocol is a natural choice as the basis for our speculative protocol, since the home node for a page is aware of all accesses made to the page by other nodes.

The *history table* is a sequence of n history entries that record the most recent **PAGE** or **DIFF** requests processed for a page, where n is the *history depth*. Each history entry records the request type (**DIFF** or **PAGE**) and the node making the request. For example, the history table in Fig. 1 indicates that the last two requests 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** requests 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 sequences of n **PAGE** or **DIFF** requests processed for a page, along with a predicted next request (labeled pred in Fig. 1). Each pattern table entry also contains a confidence level (labeled conf in Fig. 1) whose purpose is to identify patterns that may have low probability of predicting future accesses.

When the home node receives a new request for a page, it searches the pattern table for a pattern whose first n entries match the current contents of the history table. If no match is found, a new pattern table entry is created containing the history table as the sequence of n requests, the new request as the prediction, and an initial confidence value. If a match is found, the prediction in the pattern table is compared to the new request, and if they are the same, the confidence level of the pattern is increased. If the prediction and the new request do not match, the prediction is changed to the new request, and the confidence level is decreased. The confidence level is maintained within a small range, and increments or decrements beyond this range are simply reset the maximum or minimum value, respectively. After the pattern table has been updated, the new request is placed into the history table by shifting off the oldest request and inserting the new request at the end.

The pattern table is used to predict the next **PAGE** or **DIFF** request by finding a pattern in the pattern table whose initial n entries match the sequence of entries in the history table for the page. If such a pattern is found, and the pattern confidence exceeds the confidence threshold, our protocol predicts the request(s) in the $n + 1\text{st}$ entry of the pattern as the next request for the page. We set the value of the confidence threshold to be one greater than the initial confidence value, thus requiring that the pattern be observed at least twice in a row before it is used to predict a future access.

Special care must be taken to correctly track accesses to a page performed at 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. There is also no **DIFF** request created for writes performed at the home node for a page, since the protocol allows application writes at the home node to be performed directly to the shared page. 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.

Since the version of the page at the home node is generally kept current with updates from other nodes, the protocol does not trap read access to the page by the application at the home node. However, write accesses are still trapped (unless the page is in exclusive state, as described above), and these are recorded as a **PAGE** request by the speculative protocol. When a subsequent release is performed at the home node, the speculative protocol records this a **DIFF** request for the page.

Note that **PAGE** requests are the only actions that can be performed speculatively. Therefore, as an optimization, only patterns that end in a **PAGE** request entry 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 records multiple **PAGE** requests in a single request entry by storing requesting node information in a bit string rather than as node ids. The basic motivation for this change is the observation that the order of consecutive **PAGE** requests is irrelevant to the operation of the protocol, since all will receive the same data. Therefore, it is reasonable to treat any sequence of consecutive **PAGE** requests from a given set of nodes as equivalent during pattern matching, despite the actual order of the requests. Storing the requesting node ids for multiple requests in a bit string preserves the identity of the requesters, but not the order in which the requests were received. This allows one request entry to represent a collection of equivalent patterns. This not only conserves space in the history and pattern tables, it also reduces predictor training time for applications with high levels of read sharing. Lai and Falsafi use this technique in their hardware DSM memory sharing 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* and the previously described form as the *standard predictor*.

3.2 How SHRC Acts on Predictions

Once a pattern of accesses for a page has been recorded 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 synchronization operation 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 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. The home node records speculative actions 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. For example, 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 a 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 that node. If this check fails, it generates a response message that sets the confidence level for the pattern that triggered the **SPEC** request to the minimum value to inhibit future **SPEC** requests until the access pattern is reestablished. 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 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 writeable by the application, special care must be taken in updating the local copy of the page to ensure that writes made to the local copy are not lost. In this case, the node generates 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. The node also copies the page contents in the **SPEC** request to the twin so that only the local node's changes are returned to the home node.

TABLE 1
Performance of Basic Operations in the SHRC Prototype

local page fault	48 usecs
remote page miss	370 usecs

Finally, at the time the **SPEC** request arrives, the receiving 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.

4 Performance Evaluation

4.1 Methodology

To evaluate the performance benefits of SHRC, we modified an existing home-based LRC DSM system, HLRC from Rutgers University [12], to use 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 instead of VIA because UDP can be used with commodity network infrastructures (e.g., gigabit ethernet), whereas VIA requires special purpose network interface cards and switches. SHRC would also be beneficial in environments that use VIA or similar high performance networking infrastructure, but since SHRC does not require this support, we performed our evaluation in the most general setting. Since UDP does not support reliable communication, we added the necessary windowing and retransmission logic to protect against dropped packets. However, in our experiments, we increased the socket buffer sizes and queue lengths to ensure no packet losses, so that performance results are meaningful and repeatable.

Our evaluation environment consists of a cluster of 16 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 gigabit Ethernet switch. All machines are running the RedHat 7.3 Linux OS with a 2.4.18 version kernel. The performance of local and remote page faults are presented in Table 1.

We evaluate SHRC using eight shared-memory benchmark applications.[1] The applications and their relevant input parameters are summarized in Table 2. We use five applications from the SPLASH-2 Benchmark suite [13] used in the evaluation of the HLRC DSM system [12]. `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 noncontiguous-partitions version of the SPLASH-2 ocean application, which simulates large-scale ocean movements. `Radix` performs the standard radix sort algorithm on an array of integers. `water` is a molecular dynamics application that simulates the motion of water molecules in three dimensional space.

TABLE 2
Applications and Input Parameters

Application	Input Parameters
`barnes`	96K bodies, 8 timesteps
`cholesky`	input fi le tk29.O
`em3d`	80000 nodes, 15% remote, 100 timesteps
`fft`	128 x 128 x 128 array, 16 iterations
`ocean`	514 x 514 array, 60 iterations
`radix`	32 million integers, max = 16777215, radix 256
`tomcatv`	640 x 640 array, 50 iterations
`water`	4096 molecules, 12 timesteps

Two applications come from the suite of benchmarks used by Lai and Falsafi in their work on speculation in hardware DSMs [11]. `Em3d` is a shared-memory implementation of the Split-C program to perform 3D modeling of electromagnetic waves [14], 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 [15]. 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. 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 has only a marginal impact on overall runtime.

4.2 Performance Results

Fig. 2 presents the performance results for our eight benchmark applications. The figure shows application speedup for a 16-node parallel execution compared to a sequential execution with all synchronization operations compiled out. The figure presents results for the base HLRC protocol without speculation, labeled nospec, 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

1. We have also studied the benefits of our protocol for four additional benchmark applications, all from the SPLASH-2 Benchmark suite: LU factorization, raytrace, volrend, water-nsquared. Results for these applications are consistent with those reported here, but were excluded because they would provide no additional insights.

TABLE 3
Application Characteristics

Application	shared mem size	sequential exec time	barriers	lock acquires	page misses	page time	message count	message size
barnes	116 MB	175 s	65	10	25,832	12.8 s	94,957	112 MB
cholesky	40.1 MB	4.6 s	3	3,410,000	3,750	4.76 s	16,600	13.5 MB
em3d	25.7 MB	15.8 s	202	0	19,597	8.93 s	39,195	79.7 MB
fft	96 MB	56.1 s	35	0	9,149	5.47 s	19,200	33.4 MB
ocean	94.8 MB	72.4 s	1,141	182	4,610	2.39 s	14,230	22.5 MB
radix	244 MB	13.7 s	11	48	2,010	0.986 s	16,700	30.3 MB
tomcatv	11.3 MB	23 s	196	0	363	0.194 s	733	1.48 MB
water	16.1 MB	370 s	68	64	45,152	13.8 s	90,425	36.3 MB

improvements. In Section 4.4, 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. Fig. 2 shows the 95 percent confidence interval for the actual improvement, determined using a paired t-test for unequal means. For six of our eight applications, SHRC achieves a statistically significant performance improvement in all four configurations shown. However, performance is at best unchanged for radix and degrades significantly for cholesky.

For two applications, em3d and fft, performance of the application using SHRC is improved by a factor of 1.75 to 2.0 compared to the base HLRC protocol. 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.

Speculation improves performance for barnes, tomcatv, and ocean by a factor of 1.3 to 1.6 compared to the base HLRC protocol. Communication patterns in barnes are data dependent and unstructured, but remain relatively stable once established. Tomcatv is a stencil application in which processors read data produced by their nearest neighbors in a very regular pattern. Ocean is a single-writer application with coarse grain sharing. It makes heavy use of barrier synchronization and, therefore, benefits from the integration of speculative actions with barrier processing.

Speculation improves performance for water by a factor of about 1.1. In water, 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. As a result, access patterns in water are generally quite regular. While the performance improvement for water is statistically significant, is it modest in comparison to the improvement in the other applications with regular access patterns. This is caused by a negative interaction between speculation and the exclusive state

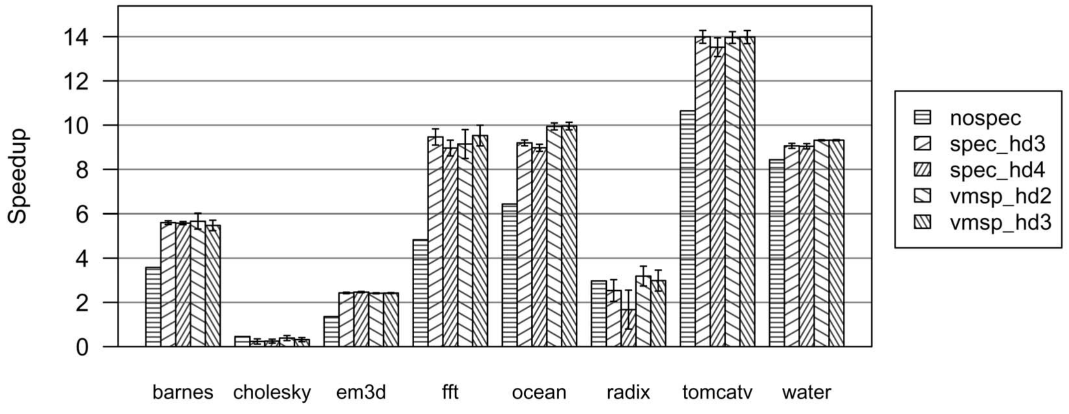

Fig. 2. Performance benefits from SHRC.

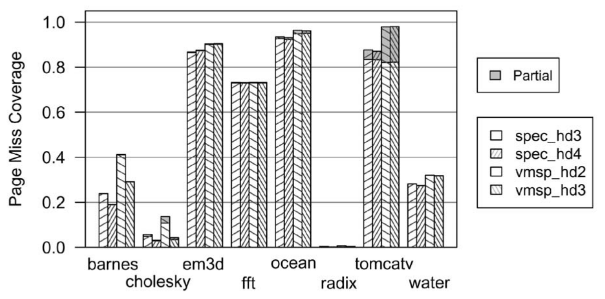

Fig. 3. Page miss coverage of the speculative protocol.

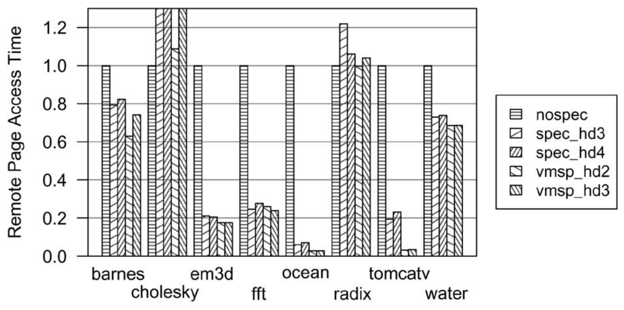

Fig. 4. Time waiting for data, normalized to base protocol.

optimization of HLRC. This is discussed in more detail in Section 4.3.

SHRC is ineffective for improving the performance of `radix` because this application makes 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, performance actually degrades 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 leads to very poor performance for `cholesky` even in the base HLRC protocol, and further degradation when using the speculative protocol.

Performance of the vector predictor configurations is marginally better than that of the standard predictors for `ocean` and `water`. This improvement is consistent, though not as large as the benefit of vector predictors for sequentially consistent hardware DSM systems [11]. We expected to see better performance for vector predictor configurations of `barnes` and `water`, which both exhibit high levels of read-sharing. For `barnes`, it appears that performance improvements for both predictors are limited by the unstructured nature of data accesses. For `water`, the performance benefits from reduced training may be limited to the first iteration of the simulation, which is not included in our performance measurements. More generally, the vector predictor may be less effective at improving performance in our system since our memory consistency model permits write sharing as well as read sharing, thus reducing the performance impact of read sharing protocol actions.

4.3 Protocol Effectiveness and Efficiency

A key measure of the effectiveness of our speculative protocol is page miss coverage, which is the ratio of correct speculative actions (those that eliminate a page miss) to the number of page misses without speculation. We estimate the number of correct speculative actions as the reduction in page misses relative to an execution without speculation. Fig. 3 presents the page miss coverage of the speculative protocol. The shaded portion of each bar represents correct speculative actions that resulted in partial page misses. The page miss coverage for `em3d`, `fft`, `ocean`, and `tomcatv` exceeds 70 percent for all configurations. These 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.

Another important measure of effectiveness is the reduction in time waiting for remote data. Fig. 4 illustrates the time waiting for remote data as a percentage of time the application spent waiting for remote data when using the base HLRC protocol. (Time waiting for data incurred by the base protocol is shown in Table 3.) This figure shows that time waiting for remote data is reduced by 70 percent to 90 percent for the four applications that achieve the greatest performance improvements from speculation. Our speculative protocol also achieves modest reductions in page access time for `barnes` and `water`. The magnitude of these reductions is consistent with the page miss coverage achieved for these two applications. As expected, there is little reduction, and in some cases even an increase, in time waiting for data in the irregular applications.

The primary metric of efficiency of our speculative protocol is accuracy, which is the percentage of speculative actions that eliminate a remote page miss. 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 configured our system to maintain the predictor history and pattern tables, and generate predictions of future accesses, but without performing any speculative requests, and then compare the predicted **PAGE** requests to the actual **PAGE** requests for each page. We then compute prediction accuracy as the ratio of the number of correct predictions to the total number of 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 of the application using the base HLRC protocol without speculation, and then compute accuracy as the ratio of correct speculative actions to all speculative actions.

Fig. 5 presents the accuracy achieved by the predictor in isolation. The predictor achieves an accuracy of 90 percent or higher for five of our eight applications: `em3d`, `fft`, `ocean`, `tomcatv`, and `water`. 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

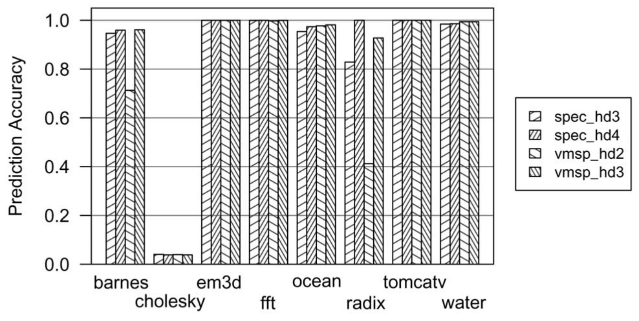

Fig. 5. Accuracy of the predictor in isolation.

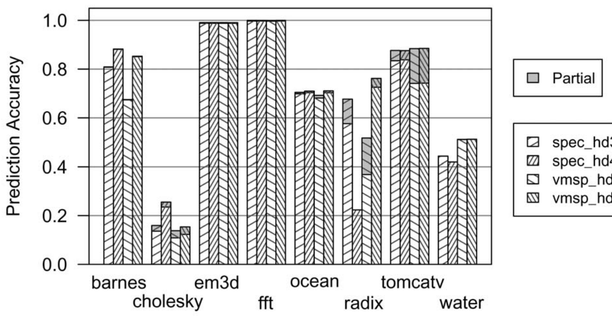

Fig. 6. Accuracy of the speculative protocol.

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 is also high, because of very regular access patterns to the water molecule structures. However, in each iteration, water 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 small fraction of incorrect predictions experienced by water.

Prediction accuracy ranges from 65 percent to 90 percent for barnes, with most predictor configurations above 85 percent accuracy. A large 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 4 percent to 8 percent 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.

Fig. 6 presents the accuracy achieved by the speculative protocol. The shaded portion of each bar indicates the portion of correct speculative actions that resulted in partial page misses. In comparison with Fig. 5, we see in Fig. 6 that accuracy of the speculative protocol is considerably lower for ocean, tomcatv, and water. We also note that tomcatv experiences a high number of partial page faults. This suggests that prediction accuracy is lower for this application because the protocol cannot execute speculative actions quickly enough to avoid the remaining page misses.

Partial page misses are quite low for ocean and water, indicating that there is another factor leading to reduced accuracy for the speculative protocol. Detailed instrumentation revealed that water 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 speculative actions cause pages in exclusive state to be transitioned to the shared state. 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, molecules are updated by the home node over a series of intervals before the updates are actually used by remote nodes. As a result, accuracy is reduced by almost 50 percent for this application.

In most applications, our speculative protocol generated very few incorrect speculative operations. However, between 6 percent and 12 percent of the speculative operations for cholesky were determined to be incorrect by the receiving node, resulting in a response message to inhibit future speculative operations for the errant pattern. This explains why the speculative protocol achieves higher accuracy for cholesky than the predictor in isolation.

Up to 24 percent of speculative operations for radix were determined to be incorrect by the receiving node, but this does not lead to improved predictor accuracy, probably because of extremely low page miss coverage, as described above. In the remaining applications, less than 0.1 percent of speculative operations were found to be incorrect by the receiving node for all configurations.

4.4 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 prior accesses used in predicting a subsequent access for a page. Fig. 7 illustrates the effect of increasing standard predictor history depth on application performance, prediction accuracy, and miss coverage for two of our benchmark applications,

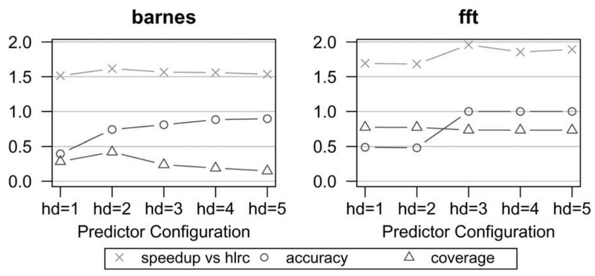

Fig. 7. Effect of standard predictor history depth.

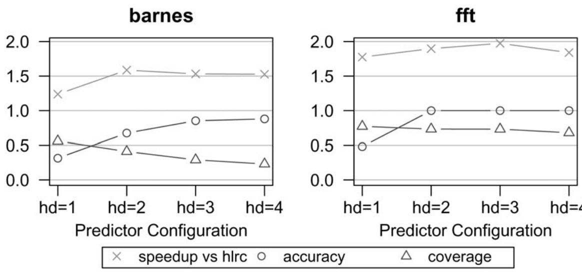

Fig. 8. Effect of vector predictor history depth.

`barnes` and `fft`. These applications are shown because they are the most sensitive to the value of history depth. We use the speedup obtained by SHRC relative to the base HLRC protocol as our measure of application performance.

In general, as history depth increases, we expect predictor accuracy to improve since predictions are based on more information. At the same time, we expect coverage to decrease 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. For `barnes`, these two effects roughly cancel each other out, and performance is relatively unaffected by value of history depth in the range we studied. 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. For `fft`, this increase in accuracy does lead to improved performance.

Fig. 8 illustrates the effect of history depth for the vector predictor for the same two applications. The same general trends in accuracy and coverage are evident for both applications. For `barnes`, accuracy improves for larger history depths, but this effect is offset by reduced coverage for history depths greater than 2. For `fft`, the vector predictor achieves nearly perfect accuracy at a history depth of 2, so it benefits little from larger history depths. As a result, both applications perform at or near their best levels for a history depth of 2.

The other applications in our study either exhibited similar though weaker effects of history depth on accuracy, coverage, and performance, or were insensitive history depth. We conclude from this analysis that both the standard 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.

4.5 Protocol Overhead

Here, we briefly report the overheads of our speculative protocol. Fig. 9 presents the number of request and response messages sent by the DSM for each configuration, normalized to the message count of an execution using the base HLRC protocol, and broken down into **DIFF**, **PAGE**, and **SPEC** messages (either requests or responses). Message counts for the base HLRC protocol are shown in Table 3. Note that the number of **DIFF** requests is consistent for all configurations, which is expected since SHRC does not generate any additional **DIFF** requests. Speculation results in a reduced message count for five of the eight applications. This reduction in message count occurs because each successful **SPEC** message eliminates two messages, a **PAGE** request and a **PAGE** response. While a reduction in the number of messages sent is a good indication of the benefits of speculation, it is not essential for improved performance, as illustrated by the standard predictor configurations of `water`, which send roughly 6 percent more messages, but also improve performance by a factor of 1.07.

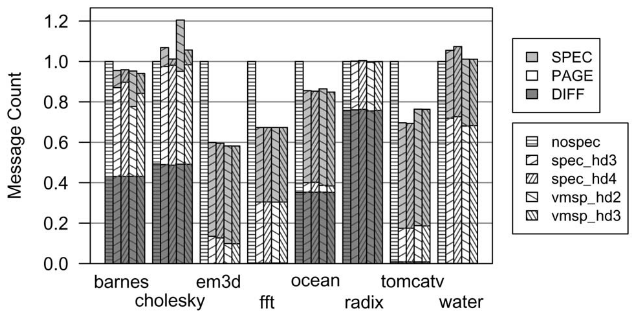

Fig. 9. Message count.

Fig. 10 presents the aggregate size of the protocol messages sent by the DSM for each configuration, normalized to aggregate message size of an execution using the base HLRC protocol, and broken down into **DIFF**, **PAGE**, and **SPEC** messages (either requests or responses). Aggregate message sizes for the base HLRC protocol are shown in Table 3. 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 percent to 90 percent 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.

In contrast, `ocean`, `tomcatv`, and `water` send 15 percent to 30 percent more data than the nonspeculative protocol. 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.

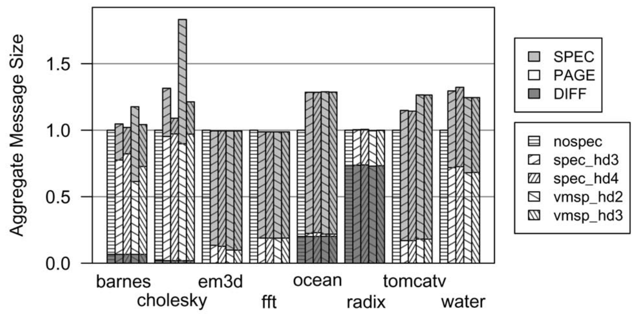

Fig. 10. Aggregate message size.

This further illustrates the importance of predictor accuracy, since the overhead of speculation increases considerably as predictor accuracy decreases. Note that performance of `tomcatv` is improved by a factor of 1.3 despite a 29 percent increase in aggregate message size. This indicates that application performance is not constrained by network bandwidth in our environment, and that speculation can improve performance by trading available network bandwidth for reduced network latency. Using netperf [16], we measured the network bandwidth that can be achieved in our environment at roughly 600 Mbps. Based on the traffic statistics from Table 3, all the applications in our study use less than 10 percent of the available bandwidth when using the base HLRC protocol.

SHRC 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 percent 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 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. With this organization, the pattern tables for most pages can be small, but can grow to accommodate a large number of patterns when necessary. 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`, the pattern tables for the standard predictor with history depth of 4 increase storage consumption by almost 1 percent. 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 0.3 percent of additional storage (for `water` with history depth of 3). These storage requirements appear quite acceptable given the performance gains that can be achieved from speculation.

4.6 Summary of Performance Results

We find that our SHRC protocol achieves statistically significant performance improvement for six of our eight benchmark applications, has high efficiency and effectiveness for iterative applications with regular access patterns, and has low storage overhead. Performance of the vector predictor configurations is marginally better than that of the standard predictors for two of our applications. In addition, the vector predictor has lower storage overhead, thus achieving its benefits at lower cost. SHRC improves application performance by converting remote page misses to local page misses, thus avoiding the latency of a remote page access. High performance network infrastructures such as VIA can reduce remote page miss latency, but do not eliminate it. Thus, SHRC should provide benefits to any environment with nonuniform page miss latencies. Further performance improvement may be possible by reducing the negative impacts of the last speculation effect, the loss of the exclusive state optimization, and partial page misses.

5 Related Work

One area of related work is the use of speculation in the context of hardware DSM systems. Lai and Falsafi developed *Memory Sharing Predictors (MSPs)*, a technique for predicting future memory accesses based on patterns of recent application/system memory reference behavior [11]. 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 percent over the base 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 hardware support for automatic updates [17], producer-initiated updates [18], and compiler generated prefetching [9].

A number of prefetching techniques have been proposed for LRC software DSM systems [19], [20]. Speight et al. describe Delphi [21], 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 percent 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.

Another important area of related work is in protocols that dynamically adapt to the behavior of the application to improve performance. Keleher was one of the first to consider adaptation between an update protocol, which eagerly pushes updates to nodes that may need them in the future, and an invalidate protocol, which simply sends invalidations for changed data, and defers sending of updates until they are specifically requested [22]. On an acquire, the Lazy Hybrid (LH) 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 [23]. Our protocol supports a wider range of synchronization mechanisms and more general access patterns.

Amza et al. [24] studied adaptation between single and multiple-writer protocols, dynamic aggregation of pages, and adaptation between update and invalidate style protocols. Our work is most closely related to this last form of adaptation, but uses a more sophisticated prediction technique. Amza et al. found that their update versus invalidate adaptation was the least effective of the techniques they studied. However, they never actually compared the techniques directly—only in specific combinations that leaves some question about the interpretation of the results. Furthermore, they provide no analysis of the efficiency or effectiveness of this adaptation, making meaningful comparisons to our protocol difficult.

Pinto et al. [25] studied a variety of techniques for reducing remote access latencies in software DSM systems. They evaluated the benefits of prefetching, adaptation between update and invalidate style protocols, and adaptation between single and multiple-writer protocols, individually and in several combinations. They choose between and update and invalidate protocol for each page based on a data classification technique. They found that prefetching outperformed their adaptation between update versus invalidate when used individually, but that the combination of techniques performs better than any technique used in isolation. Overall, their combination of techniques improved speedup for their set of applications by a factor of up to 2.1. Differences in applications and environment make comparisons to our work difficult, but it is significant to observe that our speculation technique alone can achieve performance improvements of up to a factor of 2.0.

Finally, the HLRC on VIA system [12] also has as its goal reducing the 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.

6 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 and uses synchronization operations inherent to the release consistency memory model to trigger these speculative actions. Our performance evaluation using a suite of eight shared-memory benchmark applications demonstrates that performance can be improved by a factor of 1.3 to 2.0 for applications with regular data access patterns. Some applications receive little or no benefit from speculation because inefficiencies caused by speculation offset the benefits of reduced remote page misses. 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

The authors would like to thank Liviu Iftode and Murali Rangarajan for making available their HLRC DSM system and the HLRC application suite. They also thank Babak Falsafi and An-Chow Lai for sharing their application suite for evaluating Memory Sharing Predictors for hardware DSMs. They 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] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, Sept. 1979.

[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," *Computer,* vol. 29, no. 2, pp. 18-28, Feb. 1996.

[3] L. Iftode, "Home-Based Shared Virtual Memory," PhD dissertation, Princeton Univ., 1998.

[4] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "Software versus Hardware Shared-Memory Implementation: A Case Study," *Proc. 21st Ann. Int'l Symp. Computer Architecture (ISCA-21),* pp. 106-117, Apr. 1994.

[5] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "Message Passing versus Distributed Shared Memory on Networks of Workstations," *Proc. Supercomputing Conf. '95,* Dec. 1995.

[6] T.-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Ann. Int'l Symp. Microarchitecture,* pp. 51-61, Nov. 1991.

[7] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *Proc. Int'l Symp. Microarchitecture,* pp. 281-290, 1997.

[8] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Trans. Computers,* vol. 44, no. 5, pp. 609-623, May 1995.

[9] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pp. 62-73, Oct. 1992.

[10] L. Lamport, "Time, Clocks, and the Ordering of Events in Distributed Systems," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1977.

[11] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM," *Proc. 26th Int'l Symp. Computer Architecture (ISCA 26),* pp. 172-183, June 1999.

[12] M. Rangarajan and L. Iftode, "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance," *Proc. Fourth Ann. Linux Conf.,* pp. 341-352, Oct. 2000.

[13] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

[14] 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," *Proc. Supercomputing,* pp. 262-273, 1993.

[15] 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 Int'l J. Supercomputer Applications,* vol. 5, no. 3, pp. 63-73, Fall 1991.

[16] Netperf Home Page, http://www.netperf.org/netperf/, 2005.

[17] L. Iftode, C. Dubnicki, E. Felten, and K. Li, "Improving Release-Consistent Shared Virtual Memory Using Automatic Update," *Proc. Second IEEE Symp. High-Performance Computer Architecture,* Feb. 1996.

[18] H. Abdel-Shafi, J. Hall, S.V. Adve, and V.S. Adve, "An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors," *Proc. Third Int'l Symp. High-Performance Computer Architecture,* pp. 204-215, Feb. 1997.

[19] R. Bianchini, R. Pinto, and C.L. Amorim, "Data Prefetching for Software DSMs," *Proc. Int'l Conf. Supercomputing,* pp. 385-392, 1998.

[20] M. Karlsson and P. Stenström, "Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared-Memory Systems," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 79-93, 1997.

[21] E. Speight and M. Burtscher, "Delphi: Prediction-Based Page Prefetching to Improve the Performane of Shared Virtual Memory Systems," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications,* June 2002.

[22] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, "An Evaluation of Software-Based Release Consistent Protocols," *J. Parallel and Distributed Computing,* vol. 29, no. 2, pp. 126-141, Oct. 1995.

[23] P. Keleher, "Update Protocols and Iterative Scientific Applications," *Proc. 12th Int'l Parallel Processing Symp. (IPPS),* Mar. 1998.

[24] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel, "Adaptive Protocols for Software Distributed Shared Memory," *Proc. IEEE,* special issue on distributed shared memory systems, vol. 87, no. 3, Mar. 1999.

[25] R. Pinto, R. Bianchini, and C. Amorim, "Comparing Latency-Tolerance Techniques for Software DSM Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 11, Nov. 2003.

Michael Kistler received the BA degree in mathematics and computer science in 1982 from Susquehanna University, and the MS degree in computer science in 1990 from Syracuse University. He is currently a senior software engineer in the IBM Austin Research Laboratory. His research interests include distributed and cluster computing, fault tolerance, and full-system simulation. He is a member of the IEEE.

Lorenzo Alvisi received the Laurea degree (summa cum laude) in physics (1987) from the University of Bologna, Italy, the MS (1994) and PhD (1996) degrees in computer science from Cornell University. He is an associate professor and faculty fellow in the Department of Computer Sciences at the University of Texas at Austin. His primary research interests are in reliable distributed computing. Dr. Alvisi is the recipient of an Alfred P. Sloan Research Fellowship, and IBM Faculty Partnership award, and a US National Science Foundation CAREER award. He is a member of the IEEE.