Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems

Yuanyuan Zhou, Liviu Iftode and Kai Li
Princeton University

Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems

Yuanyuan Zhou, Liviu Iftode and Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

This paper investigates the performance of shared virtual memory protocols on large-scale multicomputers. Using experiments on a 64-node Paragon, we show that the traditional Lazy Release Consistency (LRC) protocol does not scale well, because of the large number of messages it requires, the large amount of memory it consumes for protocol overhead data, and because of the difficulty of garbage collecting that data.

To achieve more scalable performance, we introduce and evaluate two new protocols. The first, Home-based LRC (HLRC), is based on the Automatic Update Release Consistency (AURC) protocol. Like AURC, HLRC maintains a home for each page to which all updates are propagated and from which all copies are derived. Unlike AURC, HLRC requires no specialized hardware support. We find that the use of homes provides substantial improvements in performance and scalability over LRC.

Our second protocol, called Overlapped Home-based LRC (OHLRC), takes advantage of the communication processor found on each node of the Paragon to offload some of the protocol overhead of HLRC from the critical path followed by the compute processor. We find that OHLRC provides modest improvements over HLRC. We also apply overlapping to the base LRC protocol, with similar results.

Our experiments were done using five of the Splash-2 benchmarks. We report overall execution times, as well as detailed breakdowns of elapsed time, message traffic, and memory use for each of the protocols.

1 Introduction

Shared memory is considered an attractive paradigm because it provides a simple yet effective parallel programming model. Research in the last decade shows that it is difficult to build or provide shared memory on a large-scale system. Although the hardware approach to implementing cache coherence has been shown to perform quite well, it requires a high engineering cost [25]. Shared virtual memory (SVM) [26], on the other hand, is a cost-effective method to provide the shared memory abstraction on a network of computers since it requires no special hardware support. The main problem with this approach has been its lack of scalable performance when compared with hardware cache coherence. The challenge is to reduce the overhead of the software coherence protocols and to implement efficient shared virtual memory that performs well with various applications on large-scale machines.

There are several factors that limit the performance of a shared virtual memory implementation. First, the large coherence granularity imposed by the underlying virtual memory page size induces false sharing and fragmentation for most applications. This effect contributes to a higher communication to computation ratio than that of the hardware cache coherence approach. The unit cost of communication of a shared virtual memory implementation is often higher than that of a hardware implementation. Second, synchronization primitives are relatively slow because they are implemented through explicit, synchronous messages. Third, performing the memory coherence protocol in software is expensive because the computation has to be interrupted to service both local and remote memory coherence protocol requests.

The known software approach to reducing the overhead of shared virtual memory is to employ relaxed memory consistency models such as Release Consistency (RC) and Lazy Release Consistency (LRC) [8, 22]. Both protocols allow multiple writers while avoiding false sharing, and both reduce overheads by maintaining coherence only at acquire and release synchronization points. The two differ on how eagerly they reconcile updates: the RC protocol propagates updates on release, whereas the LRC protocol postpones update propagations until the next acquire. Although previous prototypes have shown reasonably good performance for some applications on small systems [21], protocol overhead becomes substantial on large-scale systems. Our own experience shows that many applications do not speedup well using standard LRC-based shared virtual memory on a 32-node machine, and that the speedup curves go down when increasing the number of nodes to 64.

Recently, Automatic Update Release Consistency (AURC) [15] has been proposed as an alternative LRC-based protocol. The two protocols, AURC and LRC, have different update detection schemes. AURC uses a hardware mechanism called automatic update for update detection while LRC detects the updates in software using *diffs*. Further, in AURC updates are incorporated as they are produced into a master copy of the shared page whose (fixed) location is called its *home*. This makes update resolution in AURC extremely simple: a single full-page fetch from the home. On the other hand, in standard LRC collections of updates (diffs) are distributed and homeless, making update resolution more difficult to perform. Extensive performance evaluation [16] has shown that AURC outperforms standard LRC in most cases.

In this paper we propose two new home-based LRC protocols. The first, Home-based LRC (HLRC), is similar to the Automatic Update Release Consistency (AURC) protocol. Like AURC, HLRC maintains a home for each page to which all updates are propagated and from which all copies are derived. Unlike AURC, HLRC requires no specialized hardware support, using diffs as in LRC for update detection and propagation. This idea of using a home-based approach to build an all-software protocol similar to AURC was introduced in [17]. Our second protocol, called Overlapped Home-based LRC (OHLRC), takes advantage of the communication processor found on each node of the Paragon to offload some of the protocol overhead of HLRC from the critical path followed by the compute processor.

To evaluate the performance implications of home-based LRC, we have implemented four shared virtual memory protocols on a 64-node Intel Paragon multicomputer: HLRC, OHLRC, the standard LRC and an overlapped LRC. We compare the performance of the two home-based protocols with the two homeless LRC protocols using several Splash-2 benchmark programs. Our results show that the home-based protocols provide substantial improvements in performance over the homeless ones and that protocol overlapping using a communication processor further adds only modest improvements. By studying detailed time breakdowns, communication traffic, and memory requirements, we show also that the home-based protocols scale better than the homeless ones.

2 LRC, AURC, and Home-based LRC

In this section we briefly review the standard Lazy Release Consistency [22] and Automatic Update Release Consistency [15] protocols. We then describe our home-based and overlapped protocol variations.

2.1 Lazy Release Consistency

The standard LRC [22] is an all-software, page-based, multiple-writer protocol. It has been implemented and evaluated in the TreadMarks system [21]. The LRC protocol postpones updates to shared pages and uses the causal orders to obtain up-to-date versions. The main idea of the protocol is to use *timestamps*, or intervals, to establish the happen-before ordering between causal-related events. Local intervals are delimited by synchronization events. In our implementation an interval on processor P is delimited by one of the following two events: (i) processor P performs a remote acquire operation, or (ii) processor P receives a remote lock request.

Every writer locally records the changes it makes to every shared page during each interval. When a processor first writes a page within a new interval, it saves a copy of the page, called a *twin*, before writing to it. When the interval ends, the processor saves the interval and the page numbers that were updated in a record called a *write-notice*. The processor then compares the dirty copy of the page with the twin to detect updates and records these in a structure called a *diff*. The LRC protocol creates diffs either eagerly, at the end of each interval, or lazily, on demand.

On an acquire, the requesting processor invalidates all pages according to the write-notices received. The first access to an invalidated page causes a page fault. The page fault handler collects all the diffs for the page and applies them locally in the proper causal order to reconstitute the coherent page. Figure 1(a) shows how the protocol works with a simple example.

Such a multiple-writer protocol has several benefits. Since multiple nodes can update the same page simultaneously, the protocol can greatly reduce the protocol overhead due to false sharing. By delaying coherence actions until a synchronization point, the protocol can reduce the number of messages for protocol and data, and hence reduce software protocol overhead. Furthermore, the protocol can reduce the communication traffic due to data transfer: instead of transferring the whole page each time, the protocol transfer diffs to propagate updates.

On the other hand, diff processing can be expensive. First, the diff creations and applications all have substantial software overhead and pollute the processor's caches. Second, the acquiring processor may have to visit more than one processor to obtain diffs when multiple nodes update the same page simultaneously. Third, even when consecutive diffs of the same page (from multiple synchronization intervals) are obtained from one place, they have to be obtained as separate diffs and applied individually by the faulting processor.

The data structures of this protocol can consume a substantial amount of memory. The memory required to store diffs and write notices can grow quickly since they cannot be discarded as long as there are nodes that may still need them. When implementing the protocol on a large-scale machine, memory consumption can become a severe problem. To reduce memory consumption the shared virtual memory system must perform garbage collection frequently [21].

2.2 Automatic Update Release Consistency

The AURC protocol [16] implements Lazy Release Consistency without using any diff operations by taking advantage of the SHRIMP multicomputer's automatic update hardware mechanism [5, 6]. Automatic update provides write-through communication between local

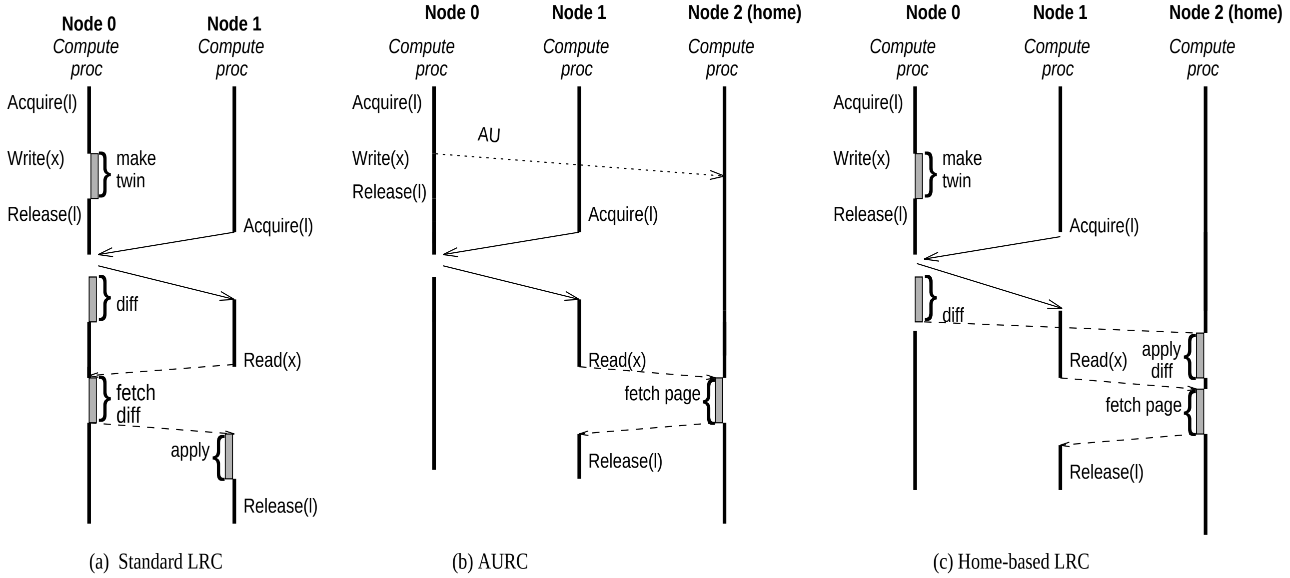


Figure 1: An example of LRC, AURC and Home-based LRC.

memory and remote memory with zero software overhead. Writes to local memory are snooped off the memory bus, propagated, and performed on remote memory by the virtual memory-mapped network interface. AURC uses the automatic update hardware instead of diffs to detect, propagate, and merge the writes performed by multiple writers on different copies of the same page.

The basic idea of AURC is to have a *home* copy for every shared page and to set automatic update mappings such that writes to other copies of the page are automatically propagated to the home. In this way the home copy is always kept up-to-date while the other copies will be updated on demand by fetching the home copy based on a LRC-like invalidation scheme. Vector timestamps are used to ensure memory coherence before pages are either accessed or fetched. Figure 1(b) shows how AURC works with a simple example. Further details can be found in [15, 16].

The AURC protocol has several advantages over the standard LRC protocol. First, it uses no diff operations. Second, there can be no page faults when accessing home node pages, so it can reduce the number of page faults if homes are chosen intelligently. Third, the non-home nodes can bring their pages up-to-date with a single round-trip communication (with the home node). Fourth, protocol data in memory and protocol messages (write-notices) under AURC are much smaller than under standard LRC.

On the other hand, AURC usually generates higher memory bus and communication traffic than LRC, both because of the write-through nature of the automatic update mechanism and because page misses always require whole-page transfers. It is nevertheless true that the latter can be outstripped by larger aggregate diff sizes in LRC. Overall, the major tradeoff between AURC and LRC is between bandwidth and protocol overhead.

In addition, the AURC method requires an automatic update mechanism. It cannot be implemented on a machine without the automatic update hardware support.

2.3 A Home-based LRC Protocol

We propose a new protocol called Home-based LRC (HLRC). Similar to the AURC protocol, it uses a “home” node for each shared page to collect updates from multiple writers. Unlike the AURC method, it requires no automatic update hardware support; it can be implemented on workstation clusters or multicomputers with traditional network interfaces.

The main idea in the HLRC protocol is to compute diffs at the end of an interval to detect updates and to transfer the updates as diffs to their homes. The lifetime of diffs is extremely short, both on the writer nodes and the home nodes. Writers can discard their diffs as soon as they are sent. Home nodes apply arriving diffs to their copies as soon as they arrive, and can then discard them. Later, on a page fault following a coherence invalidation, the faulting node fetches the whole page from the home node. Figure 1(c) shows an example of how HLRC works.

HLRC inherits most of the advantages of AURC over standard LRC: accesses to pages on their home nodes cause no page faults, non-home nodes can bring their shared pages up-to-date with a single round-trip message, and protocol data and messages are much smaller than under standard LRC.

There are two main differences between HLRC and AURC. The first is that the HLRC protocol delays propagating updates until the end of an interval, whereas the AURC method uses the automatic update mechanism to propagate eagerly. The second difference is that the HLRC protocol merges multiple updates to the same page as diffs in software and sends them together in a single message, whereas the AURC method may send one or more messages (depending on the behavior of the hardware update combining mechanism). Thus, the

HLRC method pays diffing overhead to detect updates and a regular message for each page, whereas the AURC method pays zero software overhead on update detection or message initiation.

2.4 Overlapped Protocols

Many parallel architectures [27, 14, 24, 29] contain dedicated communication and/or protocol processors that take over most of the overhead of performing these operations from the compute processor(s). Even though the occupancy of these processors is in general low, they cannot be used for general application computation since they are running a server polling loop in kernel mode. In addition, in most cases, the communication processor is one generation behind the compute processors.

This section describes two protocol variations, called Overlapped LRC and Overlapped Home-based LRC, that extend the use of the communication co-processors on the Paragon as protocol processors to overlap some of the SVM protocol overhead with computation.

2.4.1 Overlapped LRC

The Overlapped LRC (OLRC) protocol uses the communication co-processor for two tasks: to perform diffs and to service remote fetch requests. For the first task the compute processor asks its co-processor to compute diffs at the end of each interval for all pages that have been updated during that interval. Once performed, diffs are stored and propagated to remote nodes on demand.

The second task of the co-processor is to service remote requests for diffs and pages without interrupting the compute processor. These requests are sent by remote processors on a page access fault in order to bring their copies up-to-date according to the information previously provided by the write-notices. The co-processor sends the requested diffs if available. If a diff computation is in progress, the co-processor queues the request until the diff is ready. Full-page requests are issued only when the faulting node does not have a local copy for the page.

Other operations, like twin creation and diff application, are still performed by the compute processor. Usually, these do not expose enough overlapping potential to justify co-processor involvement. Also, the remote lock acquire operation, which requires coherence control handling, is still serviced by the main processor. We made this decision in an early stage of the implementation to keep the co-processor interface independent of any particular protocol.

Figure 2(a) shows an example of how the OLRC protocol overlaps its overhead with computation. Before performing `write(x)` on node 0, the compute processor creates a twin for the page holding `x`. Node 0 then releases the lock. Later, when node 1 tries to acquire lock l, it sends a lock request to node 0. After servicing the request, the compute processor of node 0 asks its co-processor to compute diffs. After acquiring the lock, the compute processor of node 1 invalidates the page holding `x` and continues its computation. The `read(x)` on node 1 causes a page fault, which triggers a diff request sent to node 0. The co-processor of node 0 handles the request. Finally, the compute processor of node 1 receives the diff, applies it to its local copy, and continues its computation.

The OLRC approach is similar to the standard LRC protocol. Moving the diff computation and fetching to the communication co-processor is easy. Applications could benefit from overlapping these operations with computation. At the same time, OLRC has the same drawbacks as LRC. The overhead of diff application as well as memory consumption can greatly affect application performance.

2.4.2 Overlapped HLRC

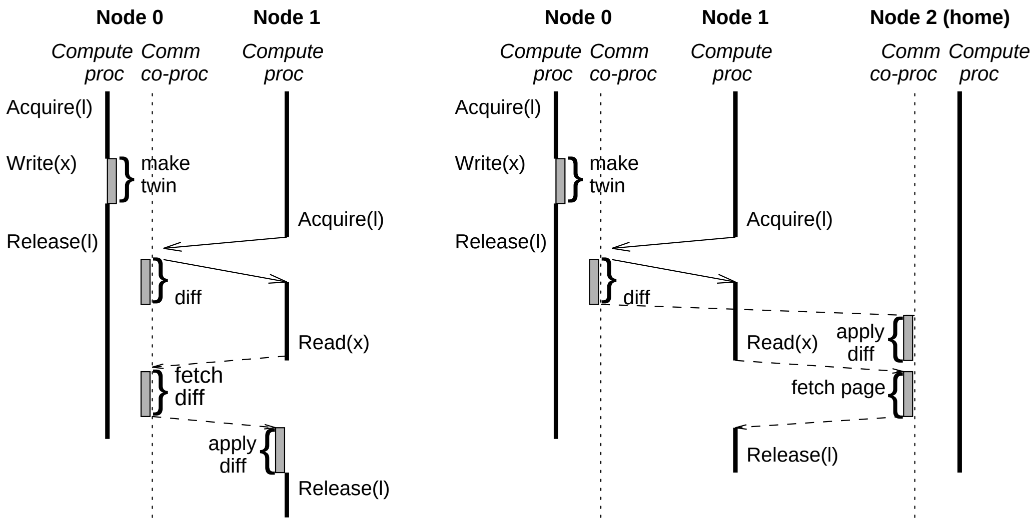
The Overlapped HLRC (OHLRC) uses the communication co-processor to perform three operations:

- Compute diffs after each interval and send them to their home node.
- Apply diffs on the local copy (at the home node).
- Service the remote requests for pages at home.

After completing a diff of a given page, the co-processor sends it to the co-processor of the home of that page. There the diff is applied and the timestamps of the corresponding page are updated. When servicing a page fetch remote request, the co-processor compares the timestamps in the request with the local timestamps of that page to ensure that the required updates are in place. If an element in the local timestamp vector is smaller than the corresponding element in the timestamp vector of the fetch request, then some diffs are still in progress and the page request is put into the pending request list attached to that page. Once all the necessary diffs have been applied, the co-processor sends the page to the faulting node.

Figure 2(b) shows an example of the OHLRC protocol. To service remote acquire requests on node 0, the current interval is ended, the co-processor starts to compute a diff, and a reply is sent back immediately to node 1, with the corresponding write notices. Node 1 invalidates the page holding `x` and then continues its computation. In the meantime, the co-processor of node 0 computes diffs and sends them to the appropriate home (node 2). The co-processor of node 2 receives the diffs and applies them to its local page. The `Read(x)` operation on node 1 causes a page fault that triggers a request sent to the home node (node 2). The co-processor of node 2 services the page request after checking the timestamps.

OLRC and OHLRC are both overlapped protocols, but their degrees of overlapping are different. Both protocols overlap diff computation and fetching with computation, but OHLRC also overlaps diff applications by performing them eagerly at the home. It appears that this may cause OHLRC to transfer more data than OLRC since OHLRC always fetches full pages from home nodes and OLRC fetches diffs instead. But this is not always true, as shown in our experiments on communication traffic. Our experiments show that OHLRC is the best protocol among the four protocols.



(a) Overlapped LRC

(b) Overlapped Home-based LRC

Figure 2: An example of Overlapped LRC and Overlapped Home-based LRC.

3 Prototype Implementations

To evaluate the performance of the protocols we have implemented four SVM prototypes on a 64-node Intel Paragon multicomputer: the standard LRC protocol, the Home-based LRC (HLRC), the Overlapped LRC (OLRC) protocol and the Overlapped Home-based LRC (OHLRC) protocol. Standard LRC is our baseline implementation. It runs on the compute processor without any overlapping. Both OLRC and OHLRC use the communication processor to overlap some protocol tasks with computation.

3.1 The Paragon System

The Intel Paragon multicomputer used in our implementation consists of 64 nodes for computation. Each node has one compute processor and a communication co-processor, sharing 64 Mbytes of local memory. Both processors are 50 MHz i860 microprocessors with 16 Kbytes of data cache and 16 Kbytes of instruction cache [18]. The data caches are coherent between the two processors. The memory bus provides a peak bandwidth of 400 MBytes/sec. The nodes are interconnected with a wormhole routed 2-D mesh network whose peak bandwidth is 200 Mbytes/sec per link [31].

The operating system is a micro-kernel based version of OSF/1 with multicomputer extensions for a parallel programming model and the NX/2 message passing primitives. The co-processor runs exclusively in kernel mode, and it is dedicated to communication. The one-way message-passing latency of a 4-byte NX/2 message on the Paragon is about 50 μsec [27]. The transfer bandwidth for large messages depends on data alignment. When data are aligned properly, the peak achievable bandwidth at the user level is 175 Mbytes/sec. Without proper alignment, the peak bandwidth is about 45 Mbytes/sec.

The operating system uses an 8 Kbyte page size for its virtual memory, though the hardware virtual memory page size is 4 Kbytes. All implementations use the `vm_protect` Mach system call to set access protection for shared pages. Access faults can be handled by either the external memory manager or the exception handling mechanism. We used the Mach exception handling mechanism for efficiency reasons.

In our prototypes for the overlapped protocols we extended the functionality of the communication co-processor with SVM protocol related operations. The standard LRC protocol is implemented exclusively at user-level using the NX/2 message library. For the overlapped protocols we modified the co-processor kernel to perform diff-related operations.

3.2 Shared Memory API

All four prototypes support the programming interface used with the Splash-2 [28] benchmark suite. This is different from the APIs supported by other software shared virtual memory systems, such as TreadMarks. The main rationale for our decision to implement the Splash-2 API is to allow programs written for a release-consistent, shared-memory multiprocessor to run on our systems without any modification.

In our implementations, all virtual address space can be shared, and global shared memory can be dynamically allocated using `G_MALLOC`. A typical program on `P` processors starts one process first that allocates and initializes global data and then spawns the other `P-1` processes. All `P` processes perform the computation in parallel and join at the end. The only synchronization primitives used in the programs are `LOCK`, `UNLOCK` and `BARRIER`.

3.3 Co-processor Interface

The communication co-processor communicates with the compute processor via cache-coherent, shared memory. For each message passing client, which can be a kernel or user process, there is a page of memory organized as a ring buffer of request slots, called a *post page*. The client process running on the compute processor uses the post page to post requests to the co-processor and receive results back.

The code on the co-processor is a dispatch loop that runs in kernel mode with interrupts disabled. This loop inspects the next active slot in each post page. When it detects a request in one of the post pages, it uses the request number as an index in the post switch table and calls the appropriate send procedure to service the request. After the request is serviced, the co-processor puts the result or error message in some location after the message.

Within the dispatch loop, the co-processor also polls the processor status registers for incoming messages. When a packet arrives, the dispatch loop reads the packet type to index into the packet switch table, and calls the appropriate receive packet procedure.

3.4 Protocol Extensions

To use the co-processor to overlap some SVM protocol operations, we extended the communication co-processor code by adding one more request type and one more packet type to its interface to support the following operations:

Compute Diff. This operation is used in both overlapped protocols. The compute processor specifies the page, page size, and the address of the twin that was previously created with the clean contents of the page. After the co-processor validates the passed parameters, it flips the "done" flag to allow the computation processor to continue, and then starts computing diffs. When the diff computation is complete, the co-processor sends the diff together with the local timestamp to the home node (in the OHLRC case) or just saves the address of the diff in the corresponding write notice record (in the OLRC case).

Apply Diff. This operation is used in the OLRC protocol to receive new diffs. The receiving co-processor transfers the modifications to its local copy, updates the home's flush timestamp for that particular page and processor accordingly, and services pending page requests if they are satisfied by the current version of the page.

Fetch Diffs. This operation is used in the OLRC protocol to collect necessary diffs. The faulting processor submits fetch-diff requests to other nodes for one or multiple diffs. The co-processor of the destination node services the requests when the diff computation is complete.

Fetch Page. The fetch-page operation is used in both overlapped protocols. On a memory access miss, the faulting processor in the OHLRC protocol sends a page request to the co-processor of the home node with the vector of lock timestamps for that page, whereas in the OLRC protocol the page request is sent to a member in the approximate copyset for that page. The co-processor of the remote node either services this request if the page is ready, or simply puts it in a page request list. The page will be sent out when all the required diffs have arrived and have been applied.

3.5 Synchronization and Garbage Collection

Synchronization handling and related coherence checking for all four prototypes is implemented at user level using NX/2 messages. Each lock has a manager, which is assigned in a round-robin fashion among the processors. The manager keeps track of which processor has most recently requested the lock. All lock acquire requests are sent to the manager unless the node itself holds the lock. The manager forwards the lock request to the processor that last requested the lock. The request message contains the current maximal vector timestamp of the acquiring processor. When the lock arrives, it contains the releaser's knowledge of all time intervals for the requester to update its timestamp vectors.

Barriers are also implemented using a centralized manager algorithm. When a barrier is entered, each node sends the barrier manager the write notices for all intervals that the manager has not seen. The barrier manager collects the intervals from all other nodes, computes the maximal timestamp, and selectively forwards the missing write notices to each node.

For LRC and OLRC, barrier synchronizations trigger garbage collection of protocol data structures when the free memory is below a certain threshold, similar to the approach used in TreadMarks [21]. Garbage collection is quite complex because it needs to collect all "live" diffs, which are distributed on various nodes. All last writers for each individual shared page need to validate the page by requesting all the missing diffs from other nodes. The non-last writers can simply invalidate the page, and modify the copyset for that page. After this phase, the collection algorithm can clean up the heaps and data structures.

For the HLRC and OHLRC protocols, there is no need to perform garbage collection since no diffs or write notices are ever stored beyond a release or barrier.

4 Performance

This section presents performance results for our implementations on five benchmark programs. We begin with a description of the benchmark applications and problem sizes used. We then evaluate the overall performance of the four protocols on these benchmarks, using speedup on 8, 32, and 64 processors as the metric. We conclude by examining the experimental results in more detail, giving execution time breakdowns, communication traffic, and memory requirements.

4.1 Applications

To evaluate the performance of our implementations, we used two kernels (LU and SOR) and three applications (Water-Nsquared, Water-Spatial and Raytrace), four of which (LU, Water-Nsquared, Water-Spatial and Raytrace) are from the Splash-2 suite. Although the names of two of the applications (Water-Nsquared and Water-Spatial) indicate a similar problem, the algorithms and their sharing patterns are different.

Application	Problem Size	Sequential Execution Time (secs)
LU	2048×2048	1,028
SOR	1024×4096	1,360
Water Nsquared	4096 molecules	1,130
Water spatial	4096 molecules	1,080
Raytrace	Balls4.env(256x256)	956

Table 1: Benchmark applications, problem sizes, and sequential execution times.

LU performs blocked LU factorization of a dense matrix. The matrix is decomposed in contiguous blocks that are distributed to processors in contiguous chunks. Therefore this kernel exhibits coarse-grained sharing and low synchronization to computation frequency, but the computation is inherently unbalanced. The results presented in this section are for a 2048×2048 matrix with 32×32 blocks.

SOR is a kernel from the suite of programs used in TreadMarks. It corresponds to the red-black successive over-relaxation (SOR) method for solving partial differential equations. The black and red arrays are partitioned into roughly equal size bands of rows, which are distributed among the processors. Communication occurs across the boundary rows between bands and is synchronized with barriers. We ran the kernel with a 1024×4096 matrix for 51 iterations starting, as in Treadmarks, with all elements initialized randomly. We chose to run this kernel, in particular, to allow some extreme case comparison between the protocols.

Water-Nsquared simulates a system of water molecules in liquid state, using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of n molecules, and partitioned among processors into contiguous pieces of n/p molecules each. The interesting communication occurs at the end of each step when each processor updates its own n/p molecules and the following $(n/2 - n/p)$ molecules of other processors in the array, using per-partition locks to protect these updates.

Water-Spatial solves the same problem as Water-Nsquared, but uses a spatial directory rather than a brute-force method, making it more suitable for large problems. The 3-d physical space is broken up into cells, and each processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules currently within those cells. A processor reads data from those processors that own cells on the boundary of its partition. Molecules migrate slowly between cells, so the irregularity of the application, although present, has little impact on performance.

Raytrace renders complex scenes in computer graphics using an optimized ray tracing method. The accesses to the scene data, into which rays are shot in this program, are read-only and relatively uninteresting other than the fact that they cause fragmentation. The interesting communication occurs in task stealing using distributed task queues, and in updating pixels in the image plane as part of a task. Both types of access patterns are fine-grained and cause considerable false sharing and fragmentation at the page level. The original Splash-2 application was modified to reorganize the task queues and remove unnecessary synchronization to alleviate the problems observed in [16].

Table 1 shows the problem sizes and their sequential execution times. For all applications we chose relatively large problem sizes, each requiring approximately 20 minutes of sequential execution. Problem sizes were determined by the capabilities of our four prototypes: although the home-based protocols can run larger problems, we chose the largest problems runnable under all protocols and all machine sizes for the sake of comparison.

4.2 Overall Performance: Speedups

Table 2 summarizes the speedups for the LRC, HLRC, OLRC and OHLRC implementations on 8, 32 and 64 nodes. There are two key observations to be made here. First, the home-based LRC protocols(HLRC and OHLRC) clearly outperform their "homeless" counterparts (LRC and OLRC) with one exception (Raytrace on 8 node, non-overlapped protocols), in which case the speedups are comparable. These results are consistent with those obtained through simulation in the comparison between LRC and AURC [16]. Second, the performance gap between home and homeless protocols increases dramatically for 32 and 64 processors configurations. This result, which is consistent across all applications, reveals a significant difference in scalability between the two classes of protocols. For instance, the difference in speedups between HLRC and LRC for 64 processors reaches a factor of two for LU and Water Spatial, a factor of 3 for SOR and a factor of almost 6 for Raytrace. For two of these applications (Water Spatial and Raytrace) the speedups under the LRC protocol actually drop when going from 32 to 64 processors. Obviously such insights would have been impossible to guess from the 8-processor runs, where the performance of the home-based and homeless protocols are very close.

The overlapped protocols provide modest improvements over the non-overlapped ones. The range of speedup improvements varies among applications, from as little as 2-3% to as much as 30%.

Summarizing, given the limitations of the Paragon architecture (e.g., large message latency and high interrupt cost, as explained next), all five real Splash-2 applications perform surprisingly well under the home-based protocols, with more than 50% parallel efficiency on 32 nodes, and between 30% and 66% on 64 nodes.

We now turn to a more detailed examination of these results, starting with the determination of the basic operation costs on the Paragon that provide the context in which the results can be better understood.

	8 nodes				32 nodes				64 nodes			
	LRC	O LRC	H LRC	OH LRC	LRC	O LRC	H LRC	OH LRC	LRC	O LRC	H LRC	OH LRC
LU	3.9	4.6	5.2	6.0	10.0	12.8	13.5	16.6	13.0	16.0	26.0	29.4
SOR	4.3	4.8	6.4	6.5	13.0	13.6	22.7	23.6	12.3	12.6	35.7	36.9
Water Nsquared	6.7	7.0	7.0	7.1	11.7	14.0	18.9	21.0			19.2	20.6
Water spatial	6.8	7.4	7.3	7.7	14.1	17.1	20	23.5	10.6	11.5	22.6	26.4
Raytrace	7.1	7.1	7.1	7.6	10.6	10.6	26.8	28.0	7.4	7.4	40.6	43.0

Table 2: Speedups on the Intel Paragon with 8, 32, 64 nodes.

4.3 Cost of Basic Operations

Table 3 shows the costs of important basic operations on the Intel Paragon.

Operation	Time in microseconds
Message Latency	50
Page Transfer	92
Receive Interrupt	690
Twin Copy	120
Diff Creation	380-560
Diff Application	0-430
Page Fault	290
Page Invalidation	200
Page Protection	50

Table 3: Timings for basic operations on the Intel Paragon

Using the basic operation costs we can determine the minimum cost (assuming no contention) for a page miss and a lock acquire. In a non-overlapped protocol, like HLRC, a page miss takes at least 290+50+690+92+50=1,172 microseconds for a full page transfer. In an overlapped protocol, such as OHLRC, the same page miss takes only 290+50+92+50=482 microseconds. Similarly, a page miss in LRC takes at least 290+50+690+50+50=1,130 microseconds without overlapping and 440 microseconds with overlapping for one single-word diff transfer. A remote acquire request, which is intermediated by the home of the lock, costs at least 50+690+50+690+50=1,550 microseconds. This could be reduced to only 150 microseconds if this service were moved to the co-processor.

4.4 Home Effect

Table 4 shows the frequency of page faults and diff related operations for HLRC and LRC on 8 and 64-nodes. (We do not give results for the overlapped protocols because they are similar to the non-overlapped ones.) There are several "home effects" revealed by this table. First, since the home's copy of the page is eagerly updated, page faults do not occur and diffs are not created at the home. This can lead to substantial protocol savings, particularly if there is a one writer-multiple readers sharing pattern and the writer is chosen as the home. This explains why no diffs are created for LU and SOR.

The other three applications also have reduced page faults and diffs due to the home effect. However, sometimes HLRC creates slightly more diffs than LRC, as shown in the Raytrace application, because of laziness.

Finally, the home-based protocols have fewer diff-applications than the homeless ones because home-based protocols apply diffs eagerly but only once, whereas the homeless protocols may apply diffs multiple times.

4.5 Time Breakdowns

To better understand where time goes, we instrumented our systems to collect the average execution time breakdowns per node. Figure 3 shows the breakdowns, including the computation time, data transfer time, garbage collection time, synchronization time including locks and barriers, and protocol overhead. Protocol overhead includes diff and twin creation, diff application, write-notice handling, and remote request service.

We use Water Nsquared (Figure 3(c)) as an example to introduce the time breakdown analysis. In both the 8- and 32-node cases, the speedups of HLRC are better than LRC, with the advantage more significant in the 32-node case. The time breakdowns show that the differences are due to the reduction of both lock and barrier synchronization time, of data transfer time, and of protocol overhead.

Synchronization cost dominates the total overhead. To identify the sources of the execution imbalance among processors, we instrumented all applications to collect per processor execution time breakdowns between two consecutive barriers. Figure 4 shows per-processor breakdowns of the execution time for Water Nsquared between barriers 9 and 10 for both LRC and HLRC protocols on 8 and 64 processors. For 8 processors the imbalance is small and mostly due to computation imbalance. For 64 processors the computation time is evenly distributed among processors; in this case almost all the imbalance is due to lock contention and communication imbalance.

Lock waiting time is significant because page misses occur in critical sections. Therefore variations in the data transfer time are reflected in the lock waiting time as well. For instance, in Water Nsquared the lock waiting time is larger under LRC than under HLRC because the data transfer time is larger as well. Lock waiting time can cause execution imbalance due to serialization of lock acquisitions when lock contention occurs. For Water Nsquared, which is a regular application, lock contention occurs when an imbalance in data transfer time occurs. This explains why there is an imbalance in the lock waiting time for LRC but not for HLRC. For irregular applications, like Water Spatial, imbalance in

	Number of nodes nodes	Read misses		Diffs Created		Diffs Applied		Lock Acquires	Barriers
		LRC	HLRC	LRC	HLRC	LRC	HLRC		
LU	8	1,050	1,050	5,000	0	1,336	0	0	64
	64	452	452	1,396	0	1,597	0	0	64
SOR	8	343	343	25,088	0	171	0	0	98
	64	385	385	3136	0	342	0	0	98
Water Nsquared	8	938	811	1,266	810	4,809	810	652	12
	32	893	859	956	815	8,742	815	556	12
Water spatial	8	1,361	1,356	468	10	2,810	10	11	10
	64	918	878	122	66	3,948	66	11	10
Raytrace	8	339	209	551	733	3,591	733	340	1
	64	108	87	83	103	2,368	103	94	1

Table 4: Average number of operations on each node.

the lock waiting time occurs even when data transfer time is balanced.

For regular applications data transfer time imbalance occurs as a result of serialization when multiple data requests arrive at the same processor simultaneously. We call this situation a “hot spot”. Homeless protocols are likely to generate hot spots more frequently than home-based protocols because in the homeless protocols updates usually are collected from the last writer, whereas in the home-based protocols updates are distributed to homes. This situation occurs for instance in Water Nsquared, which exhibits a multiple-writer multiple-reader sharing pattern with coarse-grain read and writes [16].

The second dominating overhead is data transfer time. In all cases, the data transfer times of HLRC are smaller than those of LRC for three reasons. First, the data traffic to service the page misses is higher for the homeless protocol than for the home-based protocol. This is counter-intuitive, since LRC transfers diffs whereas HLRC always transfers full pages. However, migratory data patterns can produce aggregated diff sizes in LRC significantly larger than a page. Second, LRC has to apply the diffs on the faulting page following the happen-before partial order, whereas HLRC does not require any additional processing after fetching. Third, HLRC has fewer page misses because of the home effect.

By offloading some of the protocol overhead from the critical path to the communication processors, the overlapped protocols reduce data transfer time, protocol overhead, and synchronization time. Data transfer time is reduced because the remote fetch requests are handled on the communication processor. Synchronization cost is reduced slightly because overlapping does not change the execution imbalance among the processors. Protocol time is always reduced with overlapping, but its relative contribution to the total execution time is small.

The performance analysis for Water Nsquared holds for the other applications as well.

4.6 Communication Traffic

Table 5 shows the communication traffic of the LRC and HLRC prototypes for all applications. The traffic information gives us several insights that substantiate our time breakdown analysis. Since the overlapped protocols have approximatively the same communication traffic as the non-overlapped ones, we compare only HLRC with LRC.

For each created diff, HLRC always sends one message to update the home. In addition one message is always enough in HLRC to satisfy any page miss. On the other hand, a homeless LRC protocol requires at least one message per page miss. It requires exactly one when there is only one writer per page or when the page contains migratory data (in which case all the diffs are found at the last writer). If there is only one writer per page, as in SOR, the two protocols send about the same number of messages unless LRC is penalized by the garbage collection process, as in LU, in which case it requires additional messages to fetch pages. In a regular migratory data application, like Water Nsquared, HLRC ends up sending more messages than LRC because of the additional messages sent to update the homes eagerly. In the common case, which corresponds to multiple reader and writer pages, as in Water Spatial or Raytrace, LRC sends more messages than HLRC because it requires more than one diff message to satisfy a page miss.

The amount of update-related traffic under HLRC is substantially less than under LRC (excepting Raytrace), ranging from a factor of 1.6 to almost a factor of 10.

For applications with fine-grain sharing, like Raytrace, the comparison moves towards what we expected to be the typical comparison pattern between HLRC and LRC: more messages in LRC and more traffic in HLRC. This leads to a latency vs bandwidth tradeoff in the comparison of the two protocols with respect to data transfer time. For instance, systems like the Paragon (as well as ATM networks), which has relatively high message latency and high bandwidth, are likely to benefit more from the home-based protocols.

Finally, for the protocol traffic, the home based LRC approach is consistently cheaper than the homeless LRC protocol. The homeless protocol sends more protocol related data than a home-based one, especially for large number of processors where write notices can become substantially larger due to full vector timestamp inclusion.

In conclusion, home-based protocols scale better than the homeless protocols in terms of protocol traffic.

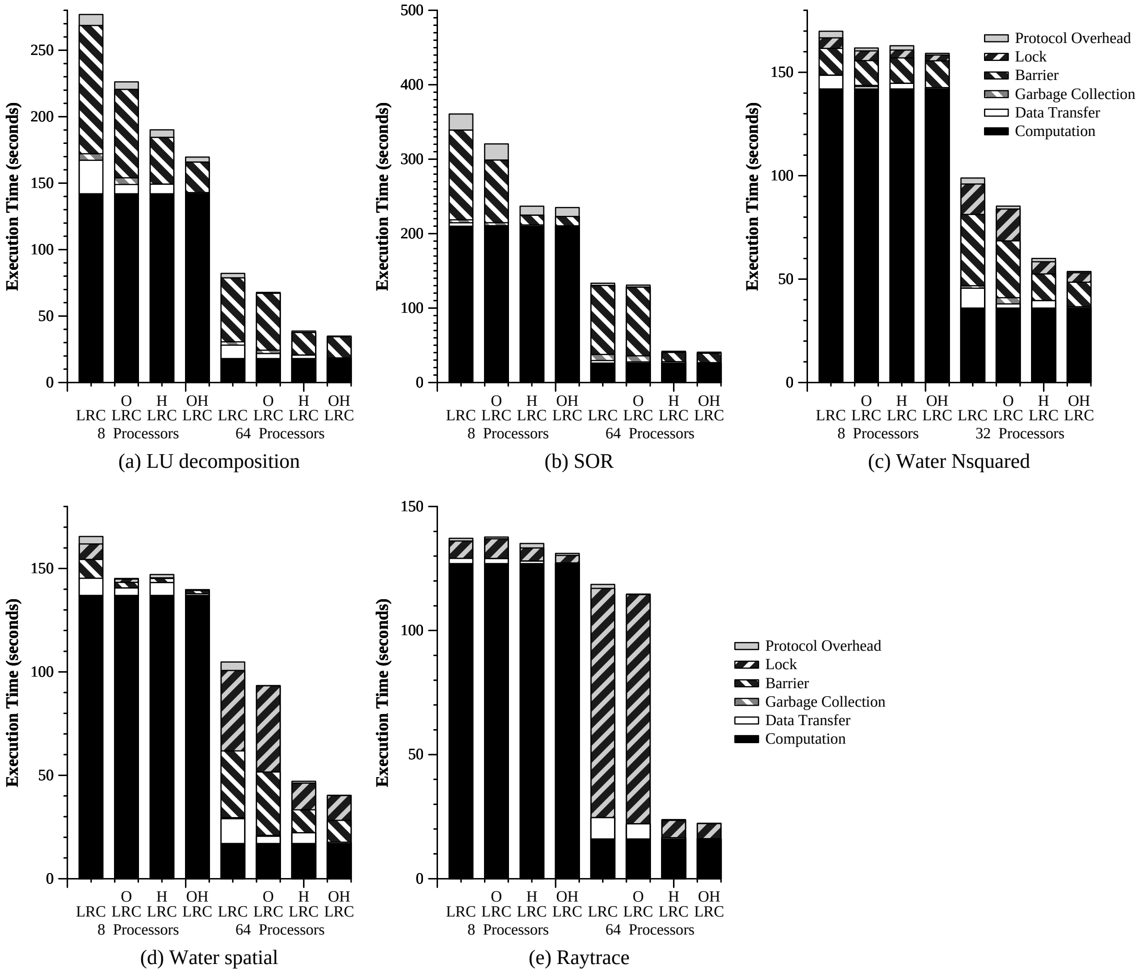


Figure 3: Time breakdowns of LU, SOR, Water Nsquared,Water Spatial and Raytrace.

4.7 Memory Requirements

Protocol memory requirement is an important criterion for scalability. It directly affects application performance in two ways: the limitation on the problem sizes and the frequency of garbage collection of protocol related data.

Table 6 reveals some interesting facts. If a garbage collection is triggered only at a barrier (as we actually implemented in our prototypes), then the memory requirement of homeless protocols can be much larger than the application memory (by a factor of 2 to 10). On the other hand, the memory used by the home-based protocols is only a small percentage of the application memory (10% to 25%). As we increase the number of processors, the memory required for protocol data tends to decrease in HLRC but to increase in LRC (except for LU). Although it is possible to reorganize the data structures to reduce the memory consumption, we do not expect that the overall picture would change dramatically.

Out of the various protocol data structures, the major memory consumers are the diffs and the write notices. In HLRC, diffs are discarded almost immediately after they are created or applied, while in LRC they have to be kept for an indefinite amount of time (until the garbage collection time in most cases). Write notices can consume a large amount of memory. For instance, the LRC protocol consumes about 5 Mbytes of memory on each of the 32 processors to store write notices in the Water-Nsquared application. Since the write-notice data structure includes the full vector timestamps in the homeless protocols, the storage requirement increases proportionally with the number of processors.

In short, home-based protocols scale much better than the homeless ones with respect to memory requirements.

4.8 Discussion

The performance evaluation we have conducted on the Paragon provides limited answers towards a complete understanding of the tradeoffs between home-based and homeless LRC protocols. The Paragon architecture is characterized by message latency, page fault, and interrupt times that are relatively large compared with memory and network bandwidth (Table 3). As a consequence, a roundtrip communication for either a page or lock transfer is at best on the order of a millisecond. Current network technologies [6, 13, 7], as well

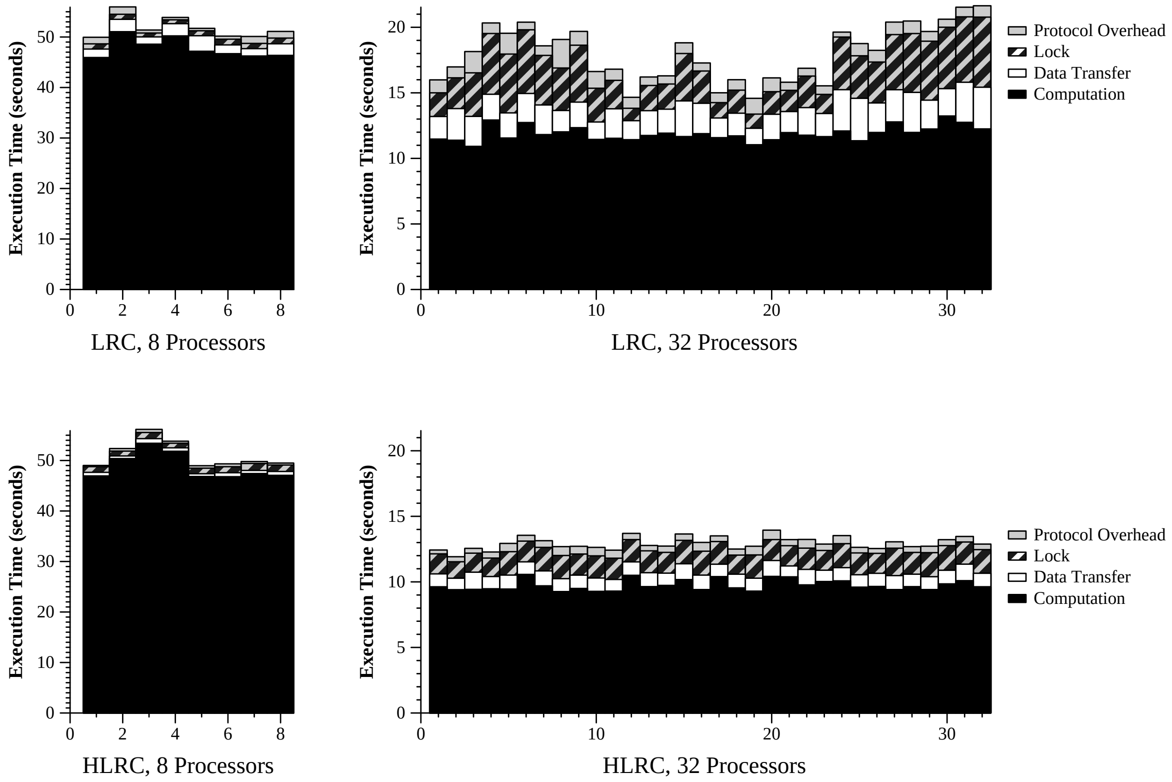


Figure 4: Time breakdowns of Water Nsquared between barriers 9 and 10.

as aggressive software for fast interrupts, exceptions [30] and virtual memory mapped communication [10, 11] have brought such latencies down significantly to the neighborhood of a couple of microseconds. An interesting question is to what extent our results are specific to the Paragon architecture and how they would be affected by different architectural parameters. Fast interrupts and low latency messages make the roundtrips shorter and since LRC has usually more messages than HLRC it is likely that the homeless protocols will benefit more from these architectural improvements. Therefore, the performance gap between the home-based and the homeless protocols would probably be smaller on such architectures.

Our performance evaluation shows that home-based protocols perform much better than the homeless ones for three regular applications with coarse-grain sharing and for two irregular applications with small communication to computation ratio. The traffic ratio for the two classes of protocols will probably be different if fine-grain sharing dominates. To help answer this question we ran SOR with all elements 0, except at the edges. Under this initialization, the interior elements of the matrix do not change for the first many iterations. Consequently, these pages have no updates during those intervals, and so do not require diffs. Even when diffs are produced later in the computation, there is only a single writer per page and only a single diff produced per interval. Consequently, the conditions of this experiment favor LRC over HLRC, which must still transfer full pages even when no updates occur. Despite these factors, the experimental results show that HLRC is still 10% better than LRC. This experiment suggests that HLRC is likely to have robust performance behavior for a large number of applications.

5 Related Work

Since shared virtual memory was first proposed ten years ago [26], a lot of work has been done on it. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence [12]. The model was used to implement shared virtual memory and reduce false sharing by allowing multiple writers [8]. Lazy Release Consistency (LRC) [22, 9, 1] further relaxed the RC protocol to reduce protocol overhead. TreadMarks [21] was the first SVM implementation using the LRC protocol on a network of stock computers. That implementation has achieved respectable performance on small-scale machines.

The recently proposed Automatic Update Release Consistency protocol (AURC) [15] is an LRC protocol that takes advantage of the automatic update mechanism in virtual memory-mapped communication. The idea of using a home-based approach to build an all-software protocol similar to AURC was proposed in [17]. Our home-based LRC protocols are based on the AURC protocol, but the updates are detected in software using diffs, as in the standard LRC. A degree of overlapping similar to the one the automatic update mechanism provides is achieved in our Overlapped Home-based LRC (OHLRC) protocol, where the communication co-processor is used to perform, transfer, and apply the diffs.

Applications	Number of nodes nodes	Update traffic				Protocol traffic			
		Number of messages		Message traffic (Mbytes)		Number of messages		Message traffic (Mbytes)	
		LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC
LU	8	1,940	1,050	29.8	8.6	2,068	1,178	0.31	0.16
	64	847	452	29.3	3.7	975	580	1.7	0.6
SOR	8	343	343	4.2	2.8	538	538	0.95	0.76
	64	385	385	6.0	3.2	581	581	2.6	0.9
Water Nsquared	8	1,102	1,621	16.1	6.8	2,784	2,531	1.8	0.8
	32	1,421	1,674	11.5	7.1	3,093	2,493	6.7	1.2
Water spatial	8	2,704	1,366	26.5	7.3	2,751	1,403	0.1	0.03
	64	2,581	944	16.4	4.7	2,628	925	0.42	0.08
Raytrace	8	1,069	942	1.46	1.43	1,446	586	0.44	0.22
	64	2,368	190	0.10	0.61	403	313	0.52	0.20

Table 5: Average communication traffic on each node.

Applications (memory used)	Number of nodes	Protocol data structure memory used (Mbytes)									
		Twins		Diffs		Page table		Write notices		Total	
		LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC
LU (32 Mbytes)	8	4.2	0	9.5	0	1.3	2.6	0.1	0.1	15.2	2.8
	64	2.0	0	6.8	0	1.3	2.6	0.7	0.2	10.8	2.8
SOR (33 Mbytes)	8	4.2	0	8.4	0	1.3	2.6	0.13	0.008	14.3	2.6
	64	2.3	0	5.2	0	1.3	2.6	0.9	0.009	9.8	2.6
Water Nsquared (2.5 Mbytes)	8	7.3	0.3	10.8	0	0.1	0.2	1.8	0.04	20.0	0.54
	32	3.9	0.1	4.2	0	0.1	0.2	5.4	0.15	13.6	0.45
Water spatial (4.8 Mbytes)	8	1.6	0.5	19.0	0	0.2	0.4	0.07	0.003	20.7	0.9
	64	0.4	0.1	24.0	0	0.2	0.4	0.4	0.01	25.0	0.5
Raytrace (10.2 Mbytes)	8	0.4	0.4	0.6	0	0.3	0.6	0.8	0.1	2.1	1.1
	64	0.7	0.1	13	0	0.3	0.7	1.3	0.1	15.2	0.9

Table 6: Average memory requirements on each node.

In a recent work [23], Keleher has shown that a simple single-writer LRC protocol perform almost as well as a more complicated multiple-writer LRC. His protocol totally eliminates diff-ing at the expense of a higher bandwidth requirement for full page transfers. Our home-based protocols support multiple writers using diffs but replace most of the diff traffic with full page traffic. The home-based protocols reduce to a single-writer protocol for applications that exhibit one-writer multiple-readers sharing patterns, like SOR or LU.

Other relaxed consistency models include Entry Consistency [3] and Scope Consistency [17]. Both models take advantage of the association of data with locks, either explicitly (Entry Consistency) or implicitly (Scope Consistency), to reduce the protocol overhead. Both Orca [2] and CRL [19] are designed to implement distributed shared memory by maintaining coherence at object level instead of page level. These methods require specialized APIs, unlike the prototype systems presented in this paper. Our systems allow programs written for a release-consistent, shared-memory multiprocessor to run without modification.

Several multicomputers use a dedicated co-processor for communication on each node. Examples include the Intel Paragon [27] and the Meiko CS-2 [14]. The Typhoon [29] system uses a special hardware board to detect access faults at fine granularity and implements distributed shared memory on a network of HyperSparc workstations. It uses one of the two CPUs in the dual-processor workstation as a protocol processor. In the Flash multiprocessor [24], each node contains a programmable processor called MAGIC that performs protocol operations and handles all communications within the node and among all nodes. Neither system uses LRC-based relaxed consistency models.

Bianchini et al. [4] proposed a dedicated protocol controller to offload some of the communication and coherence overheads from the computation processor. Using simulations they show that such a protocol processor can double the performance of TreadMarks on a 16-node configuration and that diff prefetching is not always beneficial. The protocol they evaluate is similar to our overlapped homeless LRC protocol (OLRC).

A recent study [20] investigated how to build an SVM system on a network of SMPs. They studied the tradeoffs of using a dedicated processor or the spare cycles of a compute processor to execute coherence protocol. The study is limited to simulations.

6 Limitations

In addition to the Paragon communication parameters there are several specific limitations in our experiments. The virtual memory page size used in the OSF-1 operating system is 8 Kbytes, although the hardware allows 4 Kbyte page size. We have not been able to modify the virtual memory system to change the page size to conduct more experiments. Asynchronous receives are not interruptible. The Paragon NX/2 message layer cannot support message packets larger than the page size. Therefore when the aggregate diff size exceeds the page size, the number of actual messages sent in the LRC protocol is larger than the number of logical messages.

Several possible optimizations are still unexplored. Synchronization operations have been implemented in a straightforward way using NX/2 messages and centralized algorithms. Overlapping using the communication processor was not employed for coherence and synchronization control. Finally, we have reported the results for only five applications and for a single problem size.

7 Conclusions

This paper proposes two new home-based protocols based on Lazy Release Consistency (LRC): Home-based HLRC (HLRC) and Overlapped Home-based LRC (OHLRC). Our experiments with five applications on a 64-node Intel Paragon multicomputer show that the home-based protocols perform and scale substantially better than their homeless counterparts. To our knowledge this is the first performance study of a page-based software shared memory system on such a large configuration. We have also found that protocol overlapping using the communication processor provides only modest performance improvement.

The HLRC protocol outperforms the standard LRC protocol for several reasons:

- Synchronization cost is higher for the homeless protocols. This is because the waiting time depends to a large extent on the data transfer time, which is both larger and possibly imbalanced in the homeless protocols. For all applications, the synchronization cost is the main component limiting performance for large configurations.
- Data transfer time is also larger for the homeless LRC protocols. Data traffic is larger in LRC because the aggregated diff size fetched on a page miss in LRC can exceed the size of a page, the fixed transfer size in HLRC. Also, the number of messages is usually larger in LRC. Finally, hot spots are likely to occur more frequently for a homeless protocol.
- A home-based protocol is simpler than a homeless one. As a consequence, HLRC produces less protocol overhead, generates less protocol traffic, and requires substantially less memory than LRC. Diff garbage collection is not required in HLRC because diffs have a very short lifetime in a home-based protocol.

An interesting question is whether home-based protocols are always better than homeless ones. While our study provides only limited answers, it suggests that home-based protocols are likely to perform robustly for a large number of applications.

Acknowledgments

We thank our "shepherd" John Zahorjan for his generous and thoughtful help that substantially improved this paper. This work benefitted greatly from discussions the authors had with Jaswinder Pal Singh. We are grateful to Doug Clark who encouraged us to pursue this study as well as to Czarek Dubnicki and Yuqun Chen for their aid in understanding the Paragon kernel. Pete Keleher and Sandhya Dwarkadas helped us to understand the garbage collection in TreadMarks. We also thank the anonymous reviewers for their insightful comments. Hongzhang Shan provided us with an improved version of Raytrace. We thank Matthias Blumrich for his careful reading of the draft. We want to thank Paul Messina, Heidi Lorenz-Wirzba and Sharon Brunett from the Caltech CCSC Division for providing us with the precious Paragon cycles without which this paper would not have existed.

This project is sponsored in part by ARPA under contract under grant N00014-95-1-1144 and DABT63-94-C-0049, by NSF under grant MIP-9420653, and by Intel Corporation.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. A Distributed Implementation of the Shared Data-Object Model. In *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, October 1989.

[3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.

[4] R. Bianchini, L.I Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[6] Matthias Blumrich, Cezary Dubnick, Edward Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] 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 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.

[10] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[11] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[13] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.

[14] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proceedings of Hot Interconnects*, August 1993.

[15] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[16] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[17] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[18] $i860^{TM}$ XP Microprocessor. Programmer's Reference Manual, 1991.

[19] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[20] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[21] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[22] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[23] P.J. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the IEEE COMPCON '96 Conference*, February 1996.

[24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford Flash Multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 302–313, April 1994.

[25] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, May 1992.

[26] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[27] R. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994.

[28] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[29] James R. Larus Steven K. Reinhardt and David A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[30] Ch. A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[31] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.