Lazy Release Consistency for Software Distributed Shared Memory

Pete Keleher, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University

March 9, 1992

Abstract

Relaxed memory consistency models, such as *release consistency*, were introduced in order to reduce the impact of remote memory access latency in both software and hardware distributed shared memory (DSM). However, in a software DSM, it is also important to reduce the number of messages and the amount of data exchanged for remote memory access. *Lazy release consistency* is a new algorithm for implementing release consistency that *lazily* pulls modifications across the interconnect only when necessary. Trace-driven simulation using the SPLASH benchmarks indicates that lazy release consistency reduces both the number of messages and the amount of data transferred between processors. These reductions are especially significant for programs that exhibit false sharing and make extensive use of locks.

1 Introduction

Over the past few years, researchers in hardware distributed shared memory (DSM) have proposed relaxed memory consistency models to reduce the *latency* associated with remote memory accesses [1, 7, 8, 9, 13]. For instance, in release consistency (RC) [8], writes to shared memory by processor p_1 need to be performed (become visible) at another processor p_2 only when a subsequent *release* of p_1 performs at p_2. This relaxation of the memory consistency model allows the DASH implementation of RC [11] to combat memory latency by pipelining writes to shared memory (see Figure 1). The processor is stalled only when executing a release, at which time it must wait for all its previous writes to perform.

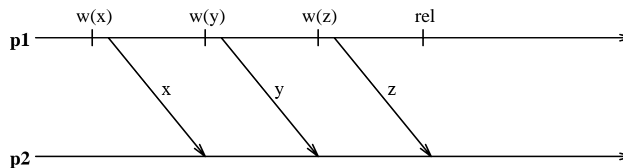

Figure 1 Pipelining Remote Memory Accesses in DASH.

In software DSMs, it is also important to reduce the *number* of messages exchanged. Sending a message in a software DSM is more expensive than in a hardware DSM, because it may involve traps into the operating system kernel, interrupts, context switches, and the execution of several layers of networking software. Ideally, the number of messages exchanged in a software DSM should equal the number of messages exchanged in a message passing implementation of the same application. Therefore, Munin's write-shared protocol [5], a software implementation of RC, buffers writes until a release, instead of pipelining them as in the DASH implementation. At the release, all writes going to the same destination are merged into a single message (see Figure 2).

Even Munin's write-shared protocol may send more messages than a message passing implementation of the

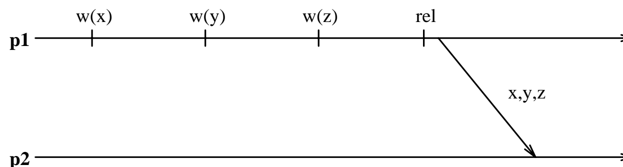

Figure 2 Merging of Remote Memory Updates in Munin.

[0]This work is supported in part by NSF Grant No. CDA-8619893 and Texas ATP Grant No. 0036404013. Pete Keleher was supported by a NASA Fellowship.

same application. Consider the example of Figure 3, where processors p_1 through p_4 repeatedly acquire the lock l, write the shared variable x, and then release l. If an update policy is used in conjunction with Munin's write-shared protocol, and x is present in all caches, then all of these cached copies are updated at every release. Logically, however, it suffices to update each processor's copy only when it acquires l. This results in a single message exchange per acquire, as in a message passing implementation. This problem is not peculiar to the use of an update policy. Similar examples can be constructed for an invalidate policy.

Lazy release consistency (LRC) is a new algorithm for implementing RC, aimed at reducing both the number of messages and the amount of data exchanged. Unlike *eager* algorithms such as Munin's write-shared protocol, *lazy* algorithms such as LRC do not make modifications globally visible at the time of a release. Instead, LRC guarantees only that a processor that acquires a lock will see all modifications that "precede" the lock acquire. The term "preceding" in this context is to be interpreted in the transitive sense: informally, a modification precedes an acquire, if it occurs before any release such that there is a chain of release-acquire operations on the same lock, ending with the current acquire (see Section 4 for a precise definition). For instance, in Figure 3, all modifications that occur in program order before any of the releases in p_1 through p_3 precede the lock acquisition in p_4. With LRC, modifications are propagated at the time of an acquire. Only the modifications that "precede" the acquire are sent to the acquiring processor. The modifications can be piggybacked on the message that grants the lock, further reducing message traffic. Figure 4 shows the message traffic in LRC for the same shared data accesses as in Figure 3. l and x are sent in a single message at each acquire.

By not propagating modifications globally at the time of the release, and by piggybacking data movement on lock transfer messages, LRC reduces both the number of messages and the amount of data exchanged. We present the results of a simulation study, using the SPLASH benchmarks, that confirms this intuition. LRC is, however, more complex to implement than eager RC because it must keep track of the "precedes" relation. We intend to implement LRC to evaluate its runtime cost. The message and data reductions seen in our simulations seem to indicate that LRC will outperform eager RC in a software DSM environment.

The outline of the rest of this paper is as follows. In Section 2, we state the definition of RC. In Section 3, we present an eager implementation of RC based on Munin's write-shared protocol. In Section 4, we define LRC and outline its implementation. In Section 5, we describe a comparison through simulation of eager RC and LRC. We briefly discuss related work in Section 6, and we draw conclusions and explore avenues for further work in Section 7.

2 Release Consistency

Release consistency (RC) [8] is a form of relaxed memory consistency that allows the effects of shared memory accesses to be delayed until certain *specially labeled* accesses occur. RC requires shared memory accesses to be labeled as either *ordinary* or *special*. Within the *special* category, accesses are divided into those labeled *sync* and *nsync*, and *sync* accesses are further subdivided into *acquires* and *releases*.

Definition 2.1 *A system is* release consistent *if:*

1. *Before an ordinary access is allowed to perform with respect to any other processor, all previous* acquires *must be performed.*
2. *Before a* release *is allowed to perform with respect to any other processor, all previous ordinary reads and writes must be performed.*
3. Special *accesses are sequentially consistent with respect to one another.*

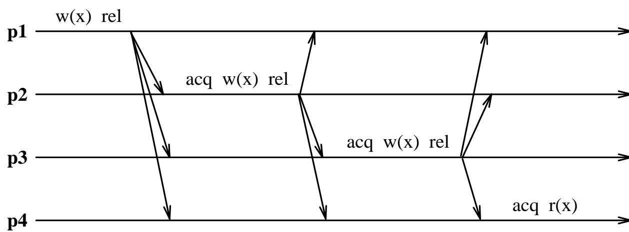

Figure 3 Repeated Updates of Cached Copies in Eager RC.

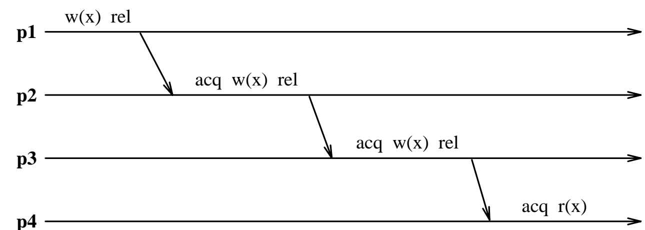

Figure 4 Message Traffic in LRC.

A write is performed with respect to another processor when reads by that processor return the new write's (or a subsequent write's) value. Reads are performed with respect to another processor when a write issued by that processor can no longer affect the value returned by the read. Accesses are *performed* when they are performed with respect to all processors in the system.

Properly labeled programs [8] produce the same results on RC memory as they would on sequentially consistent memory [10]. Informally, a program is properly labeled if there are "enough" accesses labeled as acquires or releases, such that, for all legal interleavings of accesses, each pair of conflicting ordinary accesses is separated by a release-acquire chain. Two accesses *conflict* if they reference the same memory location, and at least one of them is a write.

RC implementations can delay the effects of shared memory accesses as long as they meet the constraints of Definition 2.1.

3 Eager Release Consistency

We base our eager RC algorithm on Munin's write-shared protocol [5]. A processor delays propagating its modifications to shared data until it comes to a *release*. At that time, it propagates the modifications to all other processors that cache the modified pages. For an invalidate protocol, this simply entails sending invalidations for all modified pages to the other processors that cache these pages. In order to limit the amount of data exchanged, an update protocol sends a *diff* of each modified page to other cachers. A *diff* describes the modifications made to the page, which are then merged in the other cached copies. In either case, the release blocks until acknowledgments have been received from all other cachers.

No consistency-related operations occur on an acquire. The protocol locates the processor that last executed a release on the same variable, and the resulting value is sent from the last releaser to the current acquirer.

On an access miss, a message is sent to the directory manager for the page. The directory manager forwards the request to the current owner, and the current owner sends the page to the processor that incurred the access miss.

4 Lazy Release Consistency

In LRC, the propagation of modifications is further postponed *until the time of the acquire*. At this time, the acquiring processor determines which modifications it needs to see according to the definition of RC. To do so, LRC uses a representation of the *happened-before-1* partial order introduced by Adve and Hill [2]. The *happened-before-1* partial order is a formalization of the "preceding" relation mentioned in Section 1.

4.1 The *happened-before-1* Partial Order

We summarize here the relevant aspects of the definitions of *happened-before-1* [2].

Definition 4.1 *Shared memory accesses are partially ordered by* happened-before-1*, denoted* $\stackrel{hb1}{\rightarrow}$*, defined as follows:*

- *If a_1 and a_2 are accesses on the same processor, and a_1 occurs before a_2 in program order, then $a_1 \stackrel{hb1}{\rightarrow} a_2$.*
- *If a_1 is a release on processor p_1, and a_2 is an acquire on the same memory location on processor p_2, and a_2 returns the value written by a_1, then $a_1 \stackrel{hb1}{\rightarrow} a_2$.*
- *If $a_1 \stackrel{hb1}{\rightarrow} a_2$ and $a_2 \stackrel{hb1}{\rightarrow} a_3$, then $a_1 \stackrel{hb1}{\rightarrow} a_3$.*

RC requires that before a processor may continue past an acquire, all shared accesses that precede the acquire according to $\stackrel{hb1}{\rightarrow}$ must be performed at the acquiring processor. LRC guarantees that this property holds by propagating *write-notices* on the message that effects a release-acquire pair. A write-notice is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. Write-notices and actual values of modifications may be sent t different times in different messages.

4.2 Write-Notice Propagation

We divide the execution of each processor into distinct *intervals*, a new interval beginning with each special access executed by the processor. We define a *happens-before-1* partial order between intervals in the obvious way: an interval i_1 precedes an interval i_2 according to $\stackrel{hb1}{\rightarrow}$, if all accesses in i_1 precede all accesses in i_2 according to $\stackrel{hb1}{\rightarrow}$. An interval is said to be performed at a processor if all modifications made during that interval have been performed at that processor.

Let $V^p(i)$ be the *vector timestamp* [14] for interval i of processor p. The number of elements in the vector $V^p(i)$ is equal to the number of processors. The entry for processor p in $V^p(i)$ is equal to i. The entry for processor $q \neq p$ in $V^p(i)$ denotes the most recent interval of processor q that has performed at p.

On an acquire, the acquiring processor, p, sends its current vector timestamp V^p to the previous releaser,

q. Processor q uses this information to send p the write-notices for all intervals of all processors that have performed at q but have not yet performed at p. Releases are purely local operations in LRC, no messages are exchanged.

4.3 Data Movement Protocols

4.3.1 Multiple Writer Protocols

Both Munin and LRC allow *multiple-writer* protocols. Multiple processors can write to different parts of the same page concurrently, without intervening synchronization. This is in contrast to the exclusive-writer protocol used, for instance, in DASH [8], where a processor must obtain exclusive access to a cache line before it can be modified. Experience with Munin [5] indicates that multiple-writer protocols perform well in software DSMs, because they can handle false sharing without generating large amounts of message traffic. Given the large page sizes in software DSMs, false sharing is an important problem. Exclusive-writer protocols may cause falsely shared pages to "ping-pong" back and forth between different processors. Multiple-writer protocols allow each processor to write into a falsely shared page without any message traffic. The modifications of the different processors are later merged using the *diffs* described in Section 3.

4.3.2 Invalidate vs. Update

In the case of an invalidate protocol, the acquiring processor invalidates all pages in its cache for which it received write-notices. In the case of an update protocol, the acquiring processor updates those pages. Let i be the current interval. For each page in the cache, *diffs* must be obtained from all *concurrent last modifiers*. These are all intervals j, such that $j \stackrel{\text{hb1}}{\to} i$, the page was modified in interval j, and there is no interval k, such that $j \stackrel{\text{hb1}}{\to} k \stackrel{\text{hb1}}{\to} i$, in which the modification from interval j was overwritten.

4.3.3 Access Misses

On an access miss, a copy of the page may have to be retrieved, as well as a number of *diffs*. The modifications summarized by the *diffs* are then merged into the page before it is accessed.

On an access miss during interval i, *diffs* must be obtained for all intervals j, such that $j \stackrel{\text{hb1}}{\to} i$, the missing page was modified in interval j, and there is no interval k, such that $j \stackrel{\text{hb1}}{\to} k \stackrel{\text{hb1}}{\to} i$, in which the modification from interval j was overwritten.

If the processor still holds an (invalidated) copy of the page, LRC does not send the entire page over the interconnect. The write-notices contain all the information necessary to determine which *diffs* need to be applied to this copy of the page in order to bring it up-to-date. The *happened-before-1* partial order specifies the order in which the *diffs* need to be applied. This optimization reduces the amount of data sent.

5 Simulation

We present the results of a simulation study based on multiprocessor traces of five shared-memory application programs from the SPLASH suite [16]. We measured the number of messages and the amount of data exchanged by each program for an execution using each of four protocols: lazy update (LU), lazy invalidate (LI), eager update (EU), and eager invalidate (EI). We then relate the communication behavior to the shared memory access patterns of the application programs.

5.1 Methodology

A trace was generated from a 32-processor execution of each program using the Tango multiprocessor simulator [6]. These traces were then fed into our protocol simulator. We simulated page sizes from 512 to 8192 bytes.

We assume infinite caches and reliable FIFO communication channels. We do not assume any broadcast or multicast capability of the network.

5.2 Message Counts

The SPLASH programs use barriers and exclusive locks for synchronization. Communication occurs on barrier arrival and departure, on lock and unlock, and on an access miss. Table 1 shows the message count for each of these events under each of the protocols.

A miss costs either two or three messages for the eager protocols, depending on whether or not the directory manager has a valid copy of the page (see Section 3). For the lazy protocols, a miss requires collecting *diffs* from the *concurrent last modifiers* of the page (see Section 4.3.2).

For a lock operation, three messages are used by all four protocols for finding and transferring the lock. In addition, in LU, the new lock holder collects all the *diffs* necessary to bring its cached pages up-to-date, causing $2h$ additional messages. No extra messages are required at this time for LI, because the invalidations are piggybacked on the lock transfer message. Also, no additional messages are required for EU and EI.

On unlocks, the eager protocols send write-notices to all cachers of locally modified pages, using $2c$ messages. The lazy protocols do not communicate on unlocks.

	Access Miss	Locks	Unlocks	Barriers
LI	2m	3	0	2(n-1)
LU	2m	3+2h	0	2(n-1)+2u
EI	2 or 3	3	2c	2(n-1) + v
EU	2	3	2c	2(n-1) + 2u

m = # concurrent last modifiers for the missing page
h = # other concurrent last modifiers for any local page
c = # other cachers of the page
n = # processors in system
p = # pages in system
$u = \sum_{i=1}^{n}$(# other cachers of pages modified by i)
$v = \sum_{i=1}^{p}$(# excess invalidators of page i)

Table 1 Shared Memory Operation Message Costs

Barriers are implemented by sending an arrival message to the *barrier master* and waiting for the return of an exit message. Consequently, $2(n-1)$ messages are used to implement a barrier. In addition, both update protocols require $2u$ messages to send updates to all processors caching modified pages. The LI protocol requires no additional messages, because invalidations are piggybacked on the messages used for implementing the barrier. The EI protocol may require a small number of additional messages v to resolve multiple invalidations of a single page.

5.3 SPLASH Program Suite

5.3.1 LocusRoute

`LocusRoute` is a VLSI cell router. The major data structure is a cost grid for the cell, a cell's cost being the number of wires already running through it. Work is allocated to processors a wire at a time. Synchronization is accomplished almost entirely through locks that protect access to a central task queue.

Data movement in `LocusRoute` is largely migratory [17]: locks dominate the synchronization, and data moves according to lock accesses. As page size increases, false sharing also becomes important. Both of these factors favor lazy protocols.

Figures 5 and 6 show `LocusRoute`'s performance. The lazy protocols reduce the number of messages and the amount of data exchanged, for all page sizes.

5.3.2 Cholesky Factorization

`Cholesky` performs the symbolic and numeric portions of a `Cholesky` factorization of a sparse positive definite

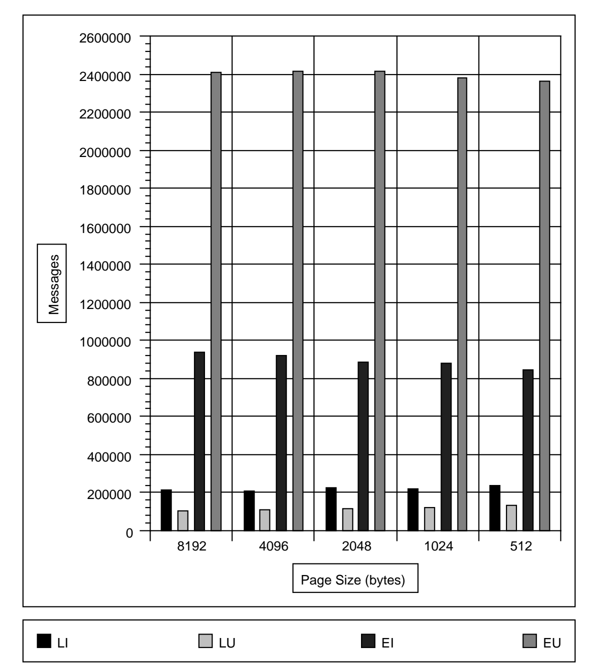

Figure 5 LocusRoute Messages.

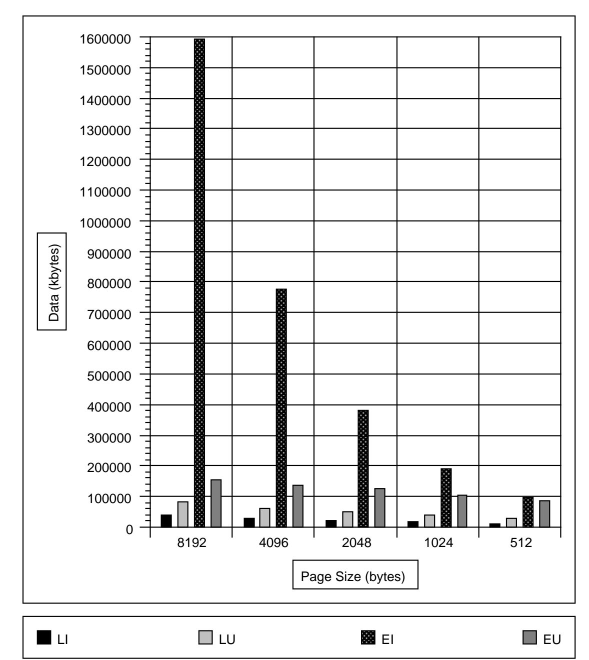

Figure 6 LocusRoute Data.

matrix. Locks are used to control access to a global task queue and to arbitrate access when simultaneous supernodal modifications attempt to modify the same column. No barriers are used.

Data motion in `Cholesky` is largely migratory, as in `LocusRoute`. The resulting performance of `Cholesky` is therefore also similar to that of `LocusRoute`: Figures 7 and 8 show that the lazy protocols reduce the number of messages and the amount of data exchanged, for all page sizes.

5.3.3 MP3D

`MP3D` simulates rarefied hypersonic airflow over an object using a Monte Carlo algorithm. Each timestep involves several barriers, with locks used to control access to global event counters.

The message traffic for `MP3D` is dominated by access misses. Figures 9 and 10 show `MP3D`'s performance. The lazy protocols exchange less data than the eager ones, because they only need to send *diffs* on an access miss and not full pages, as do the eager protocols. The update protocols exchange fewer messages, because they incur fewer access misses.

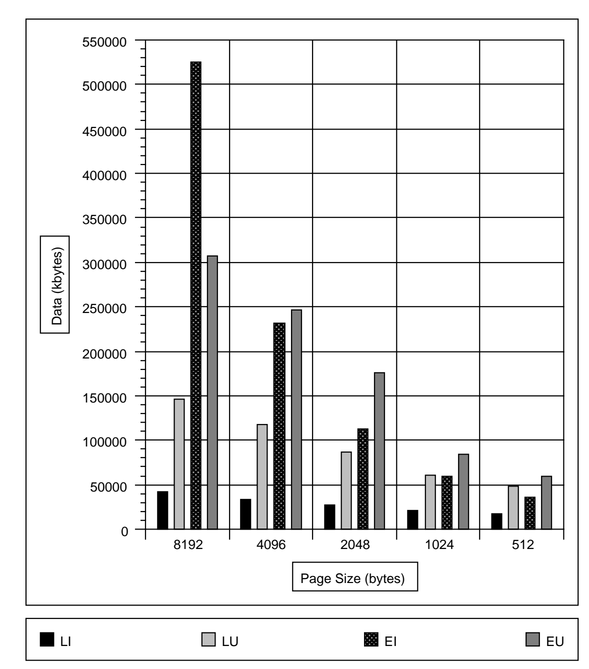

Figure 8 Cholesky Data.

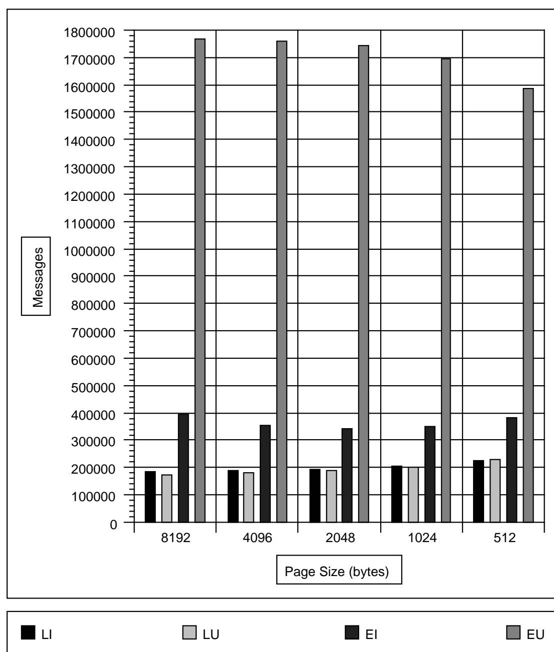

Figure 7 Cholesky Messages.

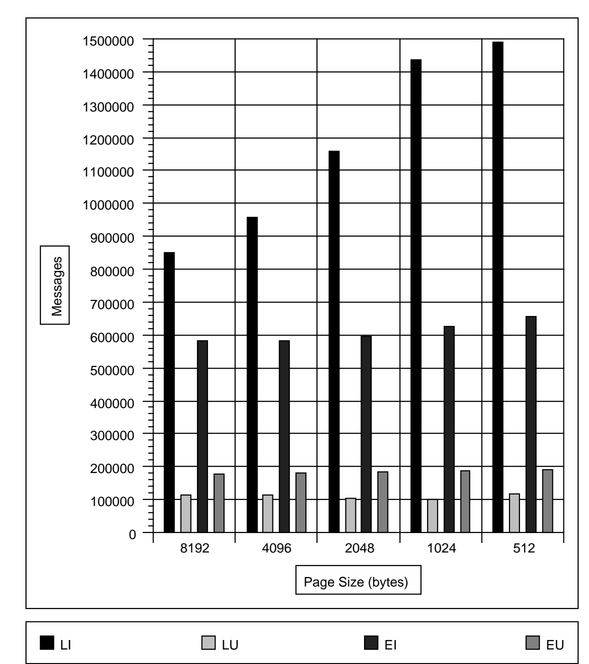

Figure 9 MP3D Messages.

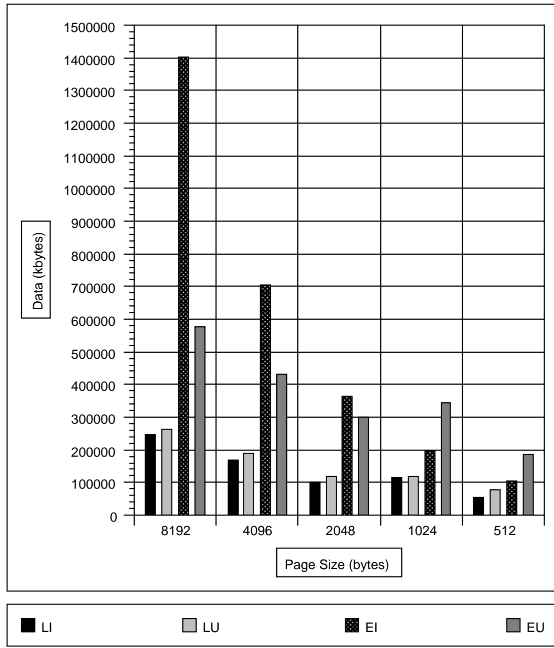

Figure 10 MP3D Data.

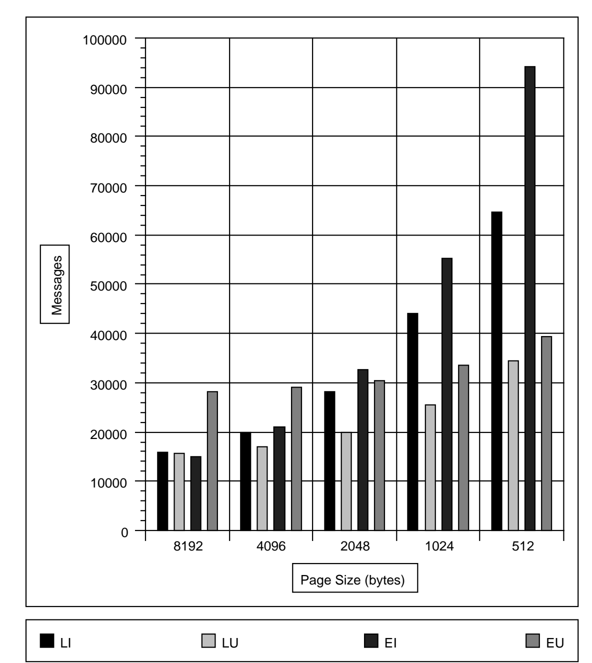

Figure 11 Water Messages.

5.3.4 Water

`Water` performs an N-body molecular dynamics simulation, evaluating forces and potentials in a system of water molecules in the liquid state. At each timestep, every molecule's velocity and potential is computed from the influences of other molecules within a spherical cut-off range. Several barriers are used to synchronize each timestep, while locks are used to control access to a global running sum and to each molecule's force sum.

Of the five benchmark programs, `Water` has the least communication. Figures 11 and 12 show the message and data traffic for `Water`. While the lazy protocols use only slightly fewer messages than eager protocols for large page sizes, their data totals are significantly lower because they can often avoid bringing an entire page across the network on an access miss.

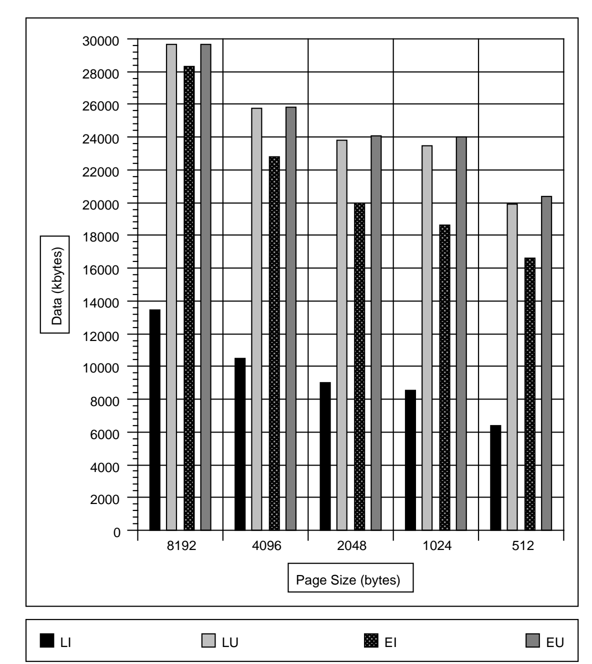

Figure 12 Water Data.

5.3.5 Pthor

`Pthor` is a parallel logic simulator. The major data structures represent logic elements, wires between elements, and per-processor work queues. Locks are used to protect access to all three types of data structures. Barriers are used only when deadlock occurs and all

task queues are empty.

In `Pthor`, each processor has a set of pages that it modifies. However, these pages are also frequently read by the other processors. Under an invalidation protocol, this causes a large number of invalidations and later reloads.

Figures 13 and 14 show `Pthor`'s performance. Data totals for EI are particularly high, because frequent reloads cause the entire page to be sent. The message count for LI is higher than for LU, because LI has more access misses.

5.4 Summary

The SPLASH programs can be divided into two categories based on their synchronization and sharing behavior. The first category is characterized by heavy use of barrier synchronization. This category includes the `MP3D` and `Water` programs. These programs performed poorly with invalidate protocols and large page sizes. Although barriers result in nearly the same number of messages under both eager and lazy protocols, even these programs have enough lock synchronization for the lazy protocols to reduce the number of messages and the amount of data exchanged.

The second category is characterized by migratory access to data that is controlled by locks. This category includes `LocusRoute`, `Cholesky` and `Pthor`. In `Cholesky` and `Pthor`, the locks protect centralized work queues, while the locks in `LocusRoute` protect access to individual cost array elements. The use of locks tends to cause the sharing patterns to closely follow synchronization. Since the lazy protocols move data according to synchronization, they handle this type of synchronization much better than eager protocols.

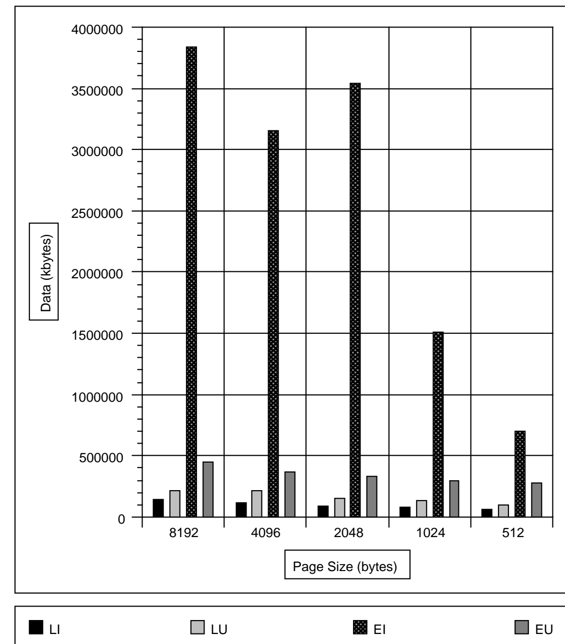

Figure 14 Pthor Data.

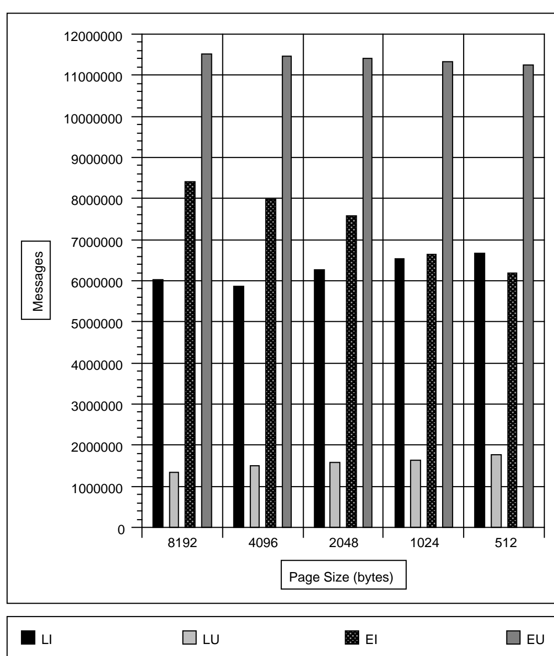

Figure 13 Pthor Messages.

LU performed well for both categories of programs. In contrast, EU often performed worse than the invalidate protocols, because it does not handle migratory data very well. LU sends fewer messages than EU for migratory data because updates are only sent to the next processor to acquire the lock that controls access to the data.

In all of the programs, the number of processors sharing a page is increased by false sharing. Multiple-writer RC protocols reduce the impact of false sharing by permitting ordinary accesses to a page by different processors to be performed concurrently. However, the eager protocols still perform communication at synchronization points between processors sharing a page, but not the data within the page. Lazy protocols eliminate this communication, because processors that falsely share data are unlikely to be causally related. This observation is consistent with the results of our simulations.

6 Related Work

Ivy [12] was the first page-based distributed shared memory system. The shared memory implemented by Ivy is sequentially consistent, and does not allow multiple writers.

Clouds [15] uses program-based *segments* rather than pages as the granularity of consistency. In addition, Clouds permits segments to be locked down at a single processor to prevent "ping-ponging".

Release consistency was introduced by Gharachorloo *et al.* [8]. It is a refinement of weak consistency, defined by Dubois and Scheurich [7]. The DASH multiprocessor takes advantage of release consistency by pipelining remote memory accesses [11]. Pipelining reduces the impact of remote memory access latency on the processor.

Munin [5] was the first software distributed shared memory system to use release consistency. Munin's implementation of release consistency merges updates at release time, rather than pipelining them, in order to reduce the number of messages transferred between processors. Munin uses multiple consistency protocols to further reduce the number of messages.

Ahamad *et al.* defined a relaxed memory model called *causal memory* [3]. Causal memory differs from RC because conflicting pairs of *ordinary* memory accesses establish causal relationships. In contrast, under RC, only *special* memory accesses establish causal relationships.

Entry consistency, defined by Bershad and Zekauskas [4], is another related relaxed memory model. EC differs from RC because it requires all shared data to be explicitly associated with some synchronization variable. As a result, when a processor acquires a synchronization variable, an EC implementation only needs to propagate the shared data associated with the synchronization variable. EC, however, requires the programmer to insert additional synchronization in shared memory programs, such as the SPLASH benchmarks, to execute correctly on an EC memory. Typically, RC does not require additional synchronization.

7 Conclusions

The performance of software DSMs is very sensitive to the number of messages and the amount of data exchanged to create the shared memory abstraction. We have described a new algorithm for implementing release consistency, *lazy release consistency*, aimed at reducing both the number of messages and the amount of data exchanged. Lazy release consistency tracks the causal dependencies between writes, acquires, and releases, allowing it to propagate writes lazily, only when they are needed.

We have used trace-driven simulation to compare lazy release consistency to an eager algorithm for implementing release consistency, based on Munin's write-shared protocol. Traces were collected from the programs in the SPLASH benchmark suite, and both update and invalidate protocols were simulated for lazy and eager RC. The simulations confirm that the number of messages and the amount of data exchanged are generally smaller for the lazy algorithm, especially for programs that exhibit false sharing and make extensive use of locks. Further work will include an implementation of lazy release consistency to assess the runtime cost of the algorithm.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical Report CS-1051, University of Wisconsin, Madison, September 1991.

[3] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, May 1991.

[4] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[6] H. Davis, S. Goldschmidt, and J. L. Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, 1990.

[7] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

[8] 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 International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[9] J.R. Goodman. Cache consistency and sequential consistency. Technical Report Technical report no. 61, SCI Committee, March 1989.

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

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] R.J. Lipton and J.S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[14] F. Mattern. Virtual time and global states of distributed systems. In Michel Cosnard, Yves Robert, Patrice Quinton, and Michel Raynal, editors, *Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers, Amsterdam, 1989.

[15] U. Ramachandran, M. Ahamad, and Y.A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[16] J.P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[17] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3th Symposium on Architectural Support' for Programming Languages and Operating Systems*, pages 243–256, April 1989.