A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory

Alan L. Cox[†], Eyal de Lara[✦], Charlie Hu[†], and Willy Zwaenepoel[†]
[†]Department of Computer Science
[✦]Department of Electrical and Computer Engineering
Rice University
{alc, delara, ychu, willy}@cs.rice.edu

Abstract

In this paper, we compare the performance of two multiple-writer protocols based on lazy release consistency. In particular, we compare the performance of Princeton's home-based protocol and TreadMarks' protocol on a 32-processor platform. We found that the performance difference between the two protocols was less than 4% for four out of seven applications. For the three applications on which performance differed by more than 4%, the TreadMarks protocol performed better for two because most of their data were migratory, while the home-based protocol performed better for one. For this one application, the explicit control over the location of data provided by the home-based protocol resulted in a better distribution of communication load across the processors.

These results differ from those of a previous comparison of the two protocols. We attribute this difference to (1) a different ratio of memory to network bandwidth on our platform and (2) lazy diffing and request overlapping, two optimizations used by TreadMarks that were not used in the previous study.

1. Introduction

In this paper, we compare two page-based software distributed shared memory (DSM) protocols based on lazy release consistency. In particular, we compare the behavior of the two most popular multiple-writer protocols, Princeton's *home-based* protocol [9] and TreadMarks' *homeless* protocol [6], on a 32-processor platform. In summary, we found that the difference in performance between the protocols is small for most applications. Thus, our results differ from a previous study in which Princeton's home-based protocol significantly outperformed a homeless protocol [9].

We attribute the difference in our findings to two factors. First, our platform has a different ratio of memory to network bandwidth. This ratio influences whether it is cheaper to use diffs, which are run-length encodings of the modifications to a page, versus full pages to maintain coherence. On the platform used in the previous study, applying a large diff was more expensive than sending a full page over the network, favoring the home-based protocol. Second, lazy diffing and request overlapping, two optimizations that are possible in a homeless protocol, and are used in TreadMarks, were not used in the previous study. In this paper, we quantify the importance of these optimizations on TreadMarks' performance.

We use seven applications: Red-Black SOR, IS, Gaussian Elimination, 3D FFT, TSP, Barnes-Hut, and Water. For four of these applications, the difference in execution time between the two protocols is less than 4%. For Water, IS and Barnes-Hut the difference is greater than 4%. The TreadMarks protocol performs better for Water and IS, two applications with mostly migratory data. On the other hand, the home-based protocol performs better for Barnes-Hut. The explicit control over the location of data provided by the home-based protocol resulted in a better distribution of communication load across the processors.

We perform our experiments on a network of 32 PCs using switched 100Mbps Ethernet. On this platform the best protocol for each application achieves speedups ranging from a worst case of 7.59 for Barnes-Hut using the home-based protocol to a best case of 25.5 for Red-Black SOR using either protocol. All of the programs achieve better speedups at 16 processors than at 8 processors. Comparing the speedups at 32 processors to the speedups at 16 processors, one program, Gaussian Elimination, slows down on the PCs. In addition, our results show that without lazy diffing and request overlapping TreadMarks' performance declines by as much as 85%.

The rest of this paper is organized as follows. Section 2 provides the necessary background on Lazy Release Consistency and the TreadMarks and home-based multiple-writer protocols. Section 3 describes our methodology, including the details on the platform that we used and the applications that we ran. Section 4 presents the results of our comparison. Finally, section 5 summarizes our conclusions.

2. Background

2.1 Lazy Release Consistency

Lazy release consistency (LRC) [6] is an algorithm that implements the release consistency (RC) [4] memory model. The LRC algorithm [6] delays the propagation of modifications to a processor until that processor performs an *acquire*. An acquire marks the beginning of a critical section. Specifically, LRC insures that the memory seen by the processor after an acquire is consistent with the happened-before-1 partial order [1].

2.2 Multiple-Writer Protocols

With a multiple-writer protocol, two or more processors can simultaneously modify their copy of a (shared) page. The two most popular multiple-writer protocols that are compatible with LRC are the TreadMarks protocol *(Tmk)* [6] and the Princeton home-based protocol *(HLRC)* [9].

In both protocols modifications to (shared) pages are detected by virtual memory faults *(twinning)* and captured by comparing the page to its twin *(diffing)* [5]. The protocols differ, however, in the location where modifications are kept and in the method by which they get propagated. These differences are described in detail below.

2.2.1 Tmk. In Tmk, a diff is not created for a modified (shared) page until a processor requests that diff in order to update its copy of the page. This *lazy* creation of diffs results in greater aggregation of changes, less diffing overhead and a reduced amount of consistency information.

When a processor faults, accessing an invalid page, it examines the write notices for that page. The write notices specify which processors have modified the page. Each write notice contains a vector timestamp that specifies the time of the modification. If one write notice's vector timestamp dominates the others, then the processor that created that write notice has all of the diffs required by the faulting processor to update its copy of the page. Thus, the faulting processor can obtain those diffs with a single request message. If, however, a page doesn't have a dominant write notice, that is, two or more write notices have concurrent vector timestamps, then the faulting processor has to send a request message to each of the corresponding processors. These request messages are overlapped to reduce the time that the faulting processor will wait.

2.2.2 HLRC. In HLRC, every shared page is statically assigned a home processor by the program. At a *release*, which marks the end of a critical section, a processor immediately generates the diffs for the pages that it has modified since its last release. It then sends these diffs to their home processor(s), where they are immediately applied to the home's copy of the page. The home's copy of a page is never invalid, but it may be write protected.

When a processor faults, accessing an invalid page, it sends a request to the page's home processor. In current implementations, the home processor always responds with a complete copy of the page, instead of one or more diffs. Thus, a diff can be discarded by the creating and home processors as soon as it is applied to the home processor's copy of the page.

2.3 Comparing the Multiple-Writer Protocols

Both protocols have strengths and weaknesses when compared to each other. For migratory data, Tmk uses half as many messages, because it transfers the diff(s) directly from the last writer to next writer. (The current implementation of Tmk avoids the problem of diff accumulation through the methods described by Amza et al. [2].) For producer/consumer data, the two protocols use roughly the same number of messages. Any difference in favor of HLRC is a result of data aggregation. Any difference in favor of Tmk is typically a result of the producer and/or consumer changing. For data in a falsely shared page, that is written by multiple processors and then read by multiple processors, the difference between the protocols is the greatest. HLRC uses significantly fewer messages as the number of readers and writers increases. Specifically, for *r* readers and *w* writers, HLRC uses at most *2w+2r* messages and Tmk uses at most *2wr* messages.

Regardless of the sharing pattern, the assignment of pages to homes is extremely important to the performance of HLRC. A poor assignment can increase the number of diffs created, messages passed, and bytes transferred by an order of magnitude. On the other hand, a good assignment can have benefits that are not possible with Tmk. For example, in the case of producer/consumer sharing, assigning the page's home to the consumer eliminates any read access faults at the consumer, because the page is always valid. On the other hand, Tmk has its share of advantages: it typically (1) transfers less data because it uses diffs to update faulting processors and (2) creates fewer diffs because their creation is delayed until they are requested by a reader.

Applications	Size / Iterations	Sequential Times (in sec.)
Barnes-Hut	65536, 3	121.6
Water	1331, 10	235.6
IS	26 X 16, 10	150.5
RB SOR	8K x 4K, 20	73.3
Gauss	4096	1547.4
3D FFT	7x7x7, 10	99.2
TSP	19 cities	227.3

Table 1: Applications, input sizes, and sequential execution times.

3 Methodology

3.1 Platform

Our platform was a switched, full-duplex 100Mbps Ethernet of thirty-two 300 MHz Pentium II-based computers. Each computer has a 512K byte secondary cache and 256M bytes of memory. All of the computers were running FreeBSD 2.2.6 and communicating through UDP sockets. The round-trip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies from 178 to 272 microseconds. The time for a 32-processor barrier is 1,333 microseconds. The time to obtain a diff varies from 313 to 1,544 microseconds, depending on the size of the diff. The time to obtain a full page is 1,308 microseconds.

3.2 Applications

We used seven applications: Red-Black SOR, Gaussian Elimination, and TSP are distributed with TreadMarks; 3D FFT and IS are NAS benchmarks [3]; Barnes-Hut and Water are SPLASH benchmarks [7, 8]. Table 1 displays the input size and sequential execution time for each of the applications.

4 Results

This section has two parts. First, we compare Tmk and HLRC. Second, we quantify the effects of lazy diff creation and request overlapping on Tmk's performance.

4.1 Tmk vs. HLRC

Table 2 presents speedups for all of the applications under Tmk and HLRC. In the rest of this section, we focus on Barnes-Hut, IS, and Water, the only three applications for which the difference in execution time between the two protocols is greater than 4%.

Figures 1, 2 and 3 present the speedups, message counts, and data transferred for Barnes-Hut, IS, and Water on 8, 16, and 32 processors.

Application	Tmk			HLRC		
	8	16	32	8	16	32
Barnes-Hut	4.84	5.83	4.84	4.85	6.51	7.59
Water	5.63	9.18	11.4	5.36	8.09	9.45
IS	7.1	12.7	17.9	6.99	12.3	16.6
RB SOR	7.62	14.8	25.5	7.65	14.5	25.4
Gauss	6.43	8.98	8.32	6.35	8.80	7.98
3D FFT	4.37	8.29	15.1	4.40	8.28	15
TSP	7.41	13.2	21.2	7.36	13.3	21.1

Table 2: Speedups on 8, 16, and 32 processors for Tmk and HLRC.

4.1.1 Barnes-Hut [7] performs an N-body simulation using the hierarchical Barnes-Hut method. There are two shared data structures: a tree used to represent the recursively decomposed subdomain (cells) of the three-dimensional physical domain containing all of the particles; and an array of particles corresponding to the leaves of the tree. Every iteration rebuilds the tree on a single processor followed by a parallel force evaluation on the particles, during which most of the tree is read by all nodes. Updates to the particle array cause a high degree of false sharing. Hence, Barnes-Hut exhibits two different access patterns: the tree is written by a single processor but read by all; while the particle array is written and read by all.

In order to explain the difference in performance between the protocols, we present results for two home assignments: HLRC/all and HLRC/particle. In HLRC/all, the home assignment for the pages containing the tree and the particle array is based on a block distribution. In HLRC/particle only the particle array is distributed in block fashion, while pages containing the tree are assigned to processor 0, the processor that rebuilds the tree.

The purpose of HLRC/particle is to limit the differences between the two protocols' behavior to the multiple-writer pages. By making processor 0 the home for the tree it becomes the source for all tree updates, mimicking the behavior of Tmk, in which updates are kept by the last writer. Hence, Tmk's and HLRC/particle's handling of the producer/consumer tree data is identical, except for the difference of diffs vs. full pages. Thus, any difference in performance must result from the treatment of the falsely shared pages of the particle array.

Our results show that while HLRC/all significantly outperforms Tmk, the speedups for Tmk and HLRC/particle are nearly identical, in spite of the vast difference in message count (1.4 million vs. 130 thousand).

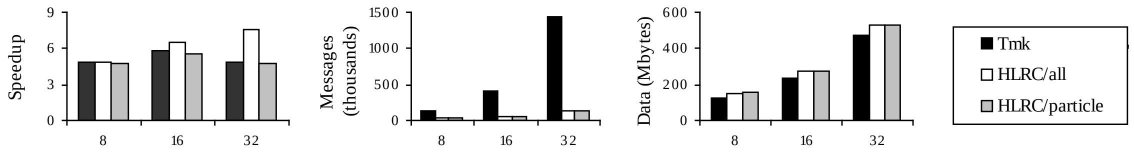

Figure 1: Speedup, messages, and data comparison among HLRC/all, HLRC/particle, and Tmk for Barnes-Hut.

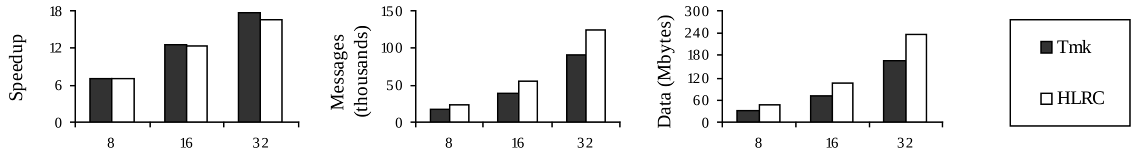

Figure 2: Speedup, messages, and data comparison among HLRC and Tmk for IS.

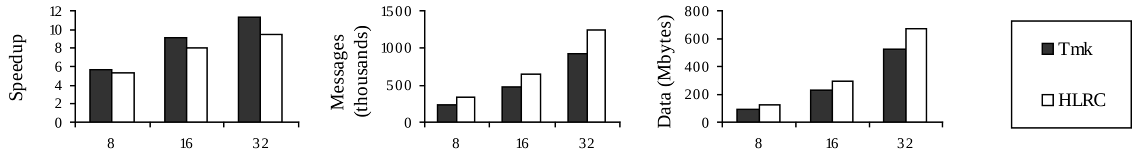

Figure 3: Speedup, messages, and data comparison among HLRC and Tmk for Water.

HLRC/particle demonstrates that HLRC's better performance does not result from a reduction in message count. Hence, the performance advantage of HLRC/all derives from its treatment of producer/consumer data (tree). The block assignment of tree pages lets HLRC/all distribute the responsibility for servicing update requests for the tree. Specifically, if the tree covers *n* pages and every processor reads the whole tree, then Tmk requires the producer of the tree to service *(p-1)*n* page requests. HLRC/all instead distributes the tree in *n*(p-1)/p* messages. After that the load of servicing the tree requests is evenly distributed.

4.1.2 IS [3] ranks a sequence of keys using a counting sort. First, processors count their keys in their private buckets. In the next phase, the values in the buckets are summed.

The sharing pattern in IS is migratory. There is no write-write false sharing, and the pages containing the shared buckets are completely overwritten by each processor. Home assignment in HLRC was done in a block fashion.

HLRC sends more messages and data than Tmk due to the migratory access pattern to the data. In HLRC, after a bucket is written, it is immediately flushed to its home processor. Since in most cases the home is not the next writer, the bucket has to be transferred a second time. In Tmk, the bucket is transferred once from the last writer to the new writer. The diff accumulation problem in Tmk is avoided through the method described by Amza et al. [2].

4.1.3 Water [8] is a molecular dynamics simulation. The main data structure in Water is a one-dimensional array of molecules. During each time step inter-molecular potentials are computed for each molecule. The parallel algorithm statically divides the array of molecules into equally large, contiguous blocks, assigning each block to a processor.

Water exhibits false sharing only for boundary pages between processors. Most of the data is shared in a migratory fashion. In HLRC, we assigned the shared molecule arrays in a block fashion.

HLRC sends more messages and data than Tmk. The reason is that it pushes diffs to their home immediately upon lock release, after a molecule is updated; whereas Tmk waits until a processor actually requests the diff. Consequently, a single diff under Tmk usually contains updates to several molecules by the time it is created. HLRC also suffers from sending whole pages, while updates of a molecule only modify about 100 bytes.

4.2 Lazy Diffing and Request Overlapping

We quantified the impact of lazy diffing and request overlapping on Tmk's performance by running the application suite on two modified versions of Tmk. In Tmk/eager, diffs are created as soon as the processor performs a release. In Tmk/sequential request overlapping is disabled. Table 3 presents the results.

Tmk/eager affected RB SOR the most. The factor of seven increase in execution time is a result of the creation of diffs for the inner rows of the band assigned to each processor. These pages are effectively private so the diffs are not used. Diff creation increased by a factor of 150. Furthermore, the amount of consistency data increased by a factor of 60.

Application	Tmk/eager			Tmk/sequential		
	8	16	32	8	16	32
Barnes-Hut	4.48	5.27	4.25	4.27	4.67	3.56
Water	5.36	fail	fail	5.6	9.09	11.1
IS	7.1	12.7	17.9	7.1	12.7	17.9
RB SOR	1.24	2.60	3.73	7.6	14.6	25.6
Gauss	1.01	1.82	2.35	6.43	8.98	8.3
3D FFT	4.37	8.29	15.1	4.15	8.08	14.8
TSP	7.41	13.1	21.4	7.36	13.2	21.1

Table 3: Effects of eager diff creation and sequential request on Tmk performance

Water would not complete on 16 and 32 processors for Tmk/eager. In both cases, eager diff creation led to a consistency message larger than the 64K byte maximum supported by UDP.

Tmk/sequential affected Barnes-Hut the most. It is the only application with significant false sharing. The disabling of request overlapping caused a 26% drop in performance.

5 Conclusions

Overall, the applications achieved speedups ranging from a worst case of 7.59 for Barnes-Hut using HLRC to a best case of 25.5 for Red-Black SOR using either protocol. All of the programs achieve better speedups at 16 processors than at 8 processors. Comparing the speedups at 32 processors to the speedups at 16 processors, one program, Gaussian Elimination, slows down. Overall, HLRC achieves from 1.6 times better speedup for Barnes-Hut to 1.2 times worse speedup for Water compared to TreadMarks.

We found the performance of the two protocols for four out of seven applications to be within 4% of each other. These results differ from a previous study where HLRC significantly outperformed a homeless protocol, like TreadMarks. We attribute the difference in our findings to two factors: a different ratio of memory to network bandwidth on our platform and lazy diffing and request overlapping, two optimizations used by TreadMarks that were not implemented in the previous study. Our results show that these optimizations are important: Without lazy diffing, RB SOR's execution time increases by a factor of seven; and without request overlapping, Barnes-Hut's execution time increases by 26%.

Barnes-Hut, IS, and Water, were the only three applications for which the difference in execution time between the two protocols is greater than 4%. TreadMarks performed better for Water and IS; two applications with migratory access patterns. HLRC performed better for Barnes-Hut. Our results show, however, that the performance advantage of HLRC does not result from its lower message count (1.4 million vs. 130 thousand). It is, instead, a result of HLRC's ability to evenly distribute, among the processors, the responsibility for providing the updates for large data structures that are produced by a single processor and consumed by multiple processors. In effect, HLRC's home assignment striped these pages across the processors, thereby spreading the load of servicing updates.

References

[1] S.V. Adve and M.D. Hill. A united formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613-624, June 1993.

[2] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. *In Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 261-271, February 1997.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[4] 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, May 1990.

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

[6] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13-21, May 1992.

[7] 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.

[8] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Computer Architecture News, 20(1):2-12, March 1992.

[9] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. *In Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75-88, November 1996.