An Efficient Lock Protocol for Home-based Lazy Release Consistency

Hee-Chul Yun   Sang-Kwon Lee   Joonwon Lee   Seungryoul Maeng

Computer Architecture Laboratory
Korea Advanced Institute of Science and Technology
373-1 Kusong-Dong, Yusong-Gu, Taejon Korea 305-701
{hcyun,sklee,joon,maeng}@camars.kaist.ac.kr

Abstract

Home-based Lazy Release Consistency (HLRC) shows poor performance on lock based applications because of two reasons: (1) a whole page is fetched on a page fault while actual modification is much smaller, and (2) a home is at the fixed location while access pattern is migratory. In this paper we present an efficient lock protocol for HLRC. In this protocol, the pages that are expected to be used by acquirer are selectively updated using diffs. The diff accumulation problem is minimized by limiting the size of diffs to be sent for each page. Our protocol reduces the number of page faults inside critical sections because pages can be updated by applying locally stored diffs. This reduction yields the reduction of average lock waiting time and the reduction of message amount. The experiment with five applications shows that our protocol archives 2% - 40% speedup against base HLRC for four applications.

1 Introduction

Research on cluster systems such as NOW (Network of Workstations) has been fuelled by the availability of powerful microprocessors and high-speed interconnection networks. Shared Virtual Memory (SVM) [9] is a cost effective solution to provide shared memory abstraction to these cluster systems.

LRC [6] and HLRC [13] are most popular protocols for SVM. Both of them are multiple-writer protocols where two or more processes can simultaneously modify their copy of a shared page. The detection of modifications is done by twinning and diffing [6]. Memory organization and diff management differentiate LRC and HLRC. In LRC, each shared page is treated like a cache. A diff is created when other process requests that diff. When a page fault occurs by accessing an invalid page, a faulting process examines the write-notices to know writers of the page and sends a diff request to them. Each process keeps diffs created by itself until a garbage collection. In HLRC, each shared page has a designated home to which all writes are propagated. Diffs are created at the end of an interval and sent to their home(s). A diff can be discarded at the creating and home process as soon as it is applied at the home process. When a page fault occurs, the faulting process fetches a fresh copy of the page from the home.

By introducing the concept of home, memory consumption for maintaining diffs and the number of messages needed to update a page is reduced. And writes to home do not produce twins and diffs. However, it has several disadvantages: home assignment is important for performance and a whole page must be fetched even when modification is small. These disadvantages severely hurt lock performance of HLRC. In general, memory area protected by lock is small and access pattern is migratory. However, in HLRC, the whole page must be fetched and home is at the fixed location. Therefore, the execution time of a critical section is increased and this can result lock serialization.

In this paper, we present an efficient lock protocol for HLRC. This protocol inherits the characteristics of HLRC but selectively updates pages that are expected to be accessed in the acquirer's critical section by sending diffs at lock release. To minimize the effect of diff accumulation problem [8], the maximum size of diffs for a page is limited to page size. If it exceeds page size, diffs for that page are not sent. Our protocol has following advantages: (1) It reduces page fault handling time inside a critical section because pages are updated at the time of lock acquire. This also helps to reduce lock-waiting time by minimizing lock serialization. (2) It reduces message amount because diffs are used to make a fresh copy instead of fetching a whole page. Our protocol was implemented in KDSM (KAIST Distributed Shared Memory) system, which is our implementation of Princeton HLRC [13]. We used five applica-

[0]This research is supported by KISTEP under the National Research Laboratory program.

tions, which include relatively many locks. For four applications, speedup is improved from 2% to 40% and the number of page faults inside critical sections is reduced 56% on the average and up to 98% against base HLRC. For one application, our protocol matched the performance with base HLRC. Overhead of our protocol is negligible.

The rest of this paper is organized as follows. Section 2 briefly introduces HLRC and shows that lock operation is inefficient in HLRC. We describe suggested protocol in section 3 and present the result of the performance comparison in section 4. Section 5 discusses related work. Finally, section 6 summarizes our conclusions.

2 Background

2.1 HLRC

HLRC is a page-based multiple writer protocol. Its prototype has been implemented at Princeton Univ [13]. Like LRC, HLRC divides program execution with *intervals*. An interval begins with each special access such as a release or an acquire. By maintaining the happened-before ordering [6] between the intervals, RC [4] can be maintained. For it, each processor has its own *vector timestamp* : the recent interval information of all processes. The main difference of HLRC against LRC is every shared page has its designated home to which all writes are propagated. To make home up-to-date, diff of each modified page is sent to its home at the end of an interval. At the time of an acquire, acquiring process receives write notices from releasing process and invalidate pages indicated by them. When an actual access happens on an invalidated page, faulting process update its stale copy by fetching the fresh copy from the home.

HLRC have several advantages over LRC [2]. First, a process can update its stale copy with one roundtrip message with home. Second, since all diffs on a shared page are collected and applied to the home copy, diffs can be discarded from the memory. Therefore, memory requirement is much smaller than LRC. Third, because home is always up-to-date, accesses at the home node do not cause any protocol action except just changing protection. However, there are also some disadvantages. First, because home is at fixed location, modification of each non-home copy must be sent to its home at the end of an interval. Therefore, if home is not wisely assigned, severe performance degradation can occur. Second, on a page fault, whole page must be fetched while modification can be much smaller.

These problems severely hurt the performance of lock operation in HLRC. In the following section, we will show the reasons.

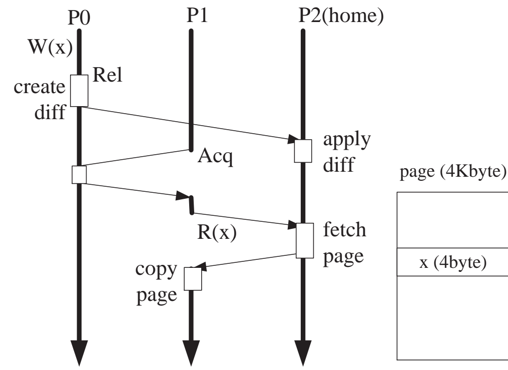

Figure 1. An example of lock operation in HLRC

2.2 Problems of using lock in HLRC

In general, memory area protected by lock is small, and its access pattern is migratory. These characteristics exactly match with the problems of HLRC. First, proper home assignment is hard due to migratory behavior of lock-protected data. Home-migration techniques [3, 10] are not helpful because they only make sense for barrier-protected data. Therefore, diff must be transferred to its home at every release. Second, a whole page must be fetched while actual modification is much smaller. This causes long latency inside a critical section thus increases the possibility of lock serialization. Lock serialization is the situation that a process is waiting for a lock, which is currently held by other process. Lock serialization badly hurt the overall performance of the system and it happens more frequently as the number of processes increase.

Figure 1 clearly shows these problems. The home process (P2) must apply received diff and serve a page request even it is neither acquirer nor releaser. Moreover, P1 fetches a page (4Kbyte), while actual modification is only 4byte size.

3 Improved Lock Protocol for HLRC

3.1 Protocol

We suggest a new lock protocol for HLRC. The main ideas of our protocol are as follows. : Releaser sends diffs for expected pages to be used by acquirer. When a page fault occurs in acquiring process, it applies received diffs for that page instead of fetching a whole page from the home. In this way, our protocol reduces page fault handling time and lock-waiting time.

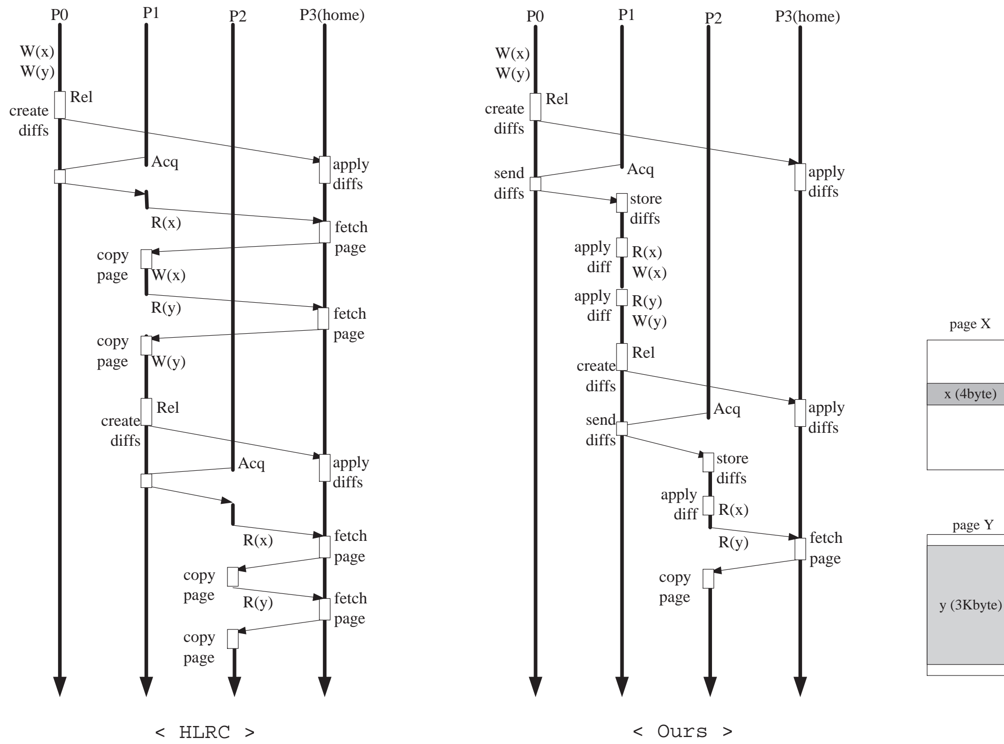

Figure 2. An example of HLRC and Our protocol

Main operations of our protocol occur at three phases: lock request, lock grant, and page fault handler. Detailed descriptions of each phase are as follows.

- Lock request
  Acquirer sends a lock request with information of expected pages to be used inside a critical section. Currently only page number information is used but version information of each page can be added to know the exact diffs that are needed to update each page. The expectation is based on previous access history of the critical section of that lock.

- Lock grant
  Releaser of that lock decides pages to send diffs based on the information from the lock request. To minimize the effect of diff accumulation problem [8], selection is based on the size of diffs to be sent for a page. If it exceeds a page size, diffs for that page are not sent. Diffs of selected pages are sent with write-notices as a lock grant message.

- Page fault handling
  To avoid diff application for unaccessed pages, received diffs are not applied immediately, but they are just stored locally. Diff application occurs at a page fault. The page fault handler examines stored diffs whether there is all diffs to make the page up-to-date. If all diffs are found, they are applied for the faulting page. If not, it fetches a fresh page from the home just as base HLRC does.

Figure 2 clearly shows the differences of our protocol from base HLRC. At the first glance, it can be noticed that page requests to home occurred only once in our protocol while base HLRC did four times. The only page request of our protocol at P2 is due to accumulated diffs. To make up-to-date copy of page Y at P2, two diffs must be sent from P1. However, summed size of these two diffs exceeds a page size. Therefore, P1 decides not to send these diffs as explained above.

3.2 The merits and demerits

Our protocol has several advantages over base HLRC.

First, it reduces a page fault handling time inside critical sections. Usually, diffs are received at the time of lock acquire. When a page fault occurred, page fault handler can make a up-to-date copy of the page by simply applying these locally stored diffs. Because diff applications take much shorter time than fetching a whole page, page fault

handing time can be greatly reduced. This reduction also helps to reduce lock-waiting time because lock serialization can be minimized. Second, it reduces message amounts. To make a page up-to-date only diffs are transferred while the whole page is transferred in base HLRC. The summed size of transferred diffs for a page is guaranteed to be smaller than a page size in our protocol. Third, extra messages for our protocol are not needed. All needed information can be piggybacked to existing protocol messages (such as lock request or lock grant message).

The overheads of our protocol are creating diffs for home pages and memory overhead to store diffs. However, these overheads can be minimized as explained in the following sections.

3.3 Home page diffing overhead and its solutions

To operate fully update manner, creating diffs for home pages is desirable. However, it is limited for the pages accessed inside critical sections. Other pages do not create diffs just as base HLRC does. In general, the number of pages accessed inside critical sections is quite small. Therefore, creating diffs for home pages do not hurt performance much. Moreover, by limiting the size of a diff, average time required to create diff for a home page can be reduced compared with for a non-home page. Because home page diffing does not affect correctness and large diffs have less possibility to be sent, this limiting helps performance. In our current implementation, this threshold value is set to 256 bytes. In this ways, overall overhead related to home page diffing is minimized.

3.4 Memory overhead and its solution

In our protocol, diff should be maintained in memory. However, it is only for the pages accessed inside critical sections as in the case of home page diffing. Therefore, memory requirement is much smaller. Moreover, because home is always updated, diffs can be discarded at anytime without affection of correctness. Therefore, complex garbage collection procedure as in LRC is not needed. Simply discarding old diffs is enough procedure for garbage collection.

4 Performance Evaluation

We performed experiments on 8 node Linux cluster which are connected by 100 Mbps Switched Fast Ethernet. Each node contains 500 MHz Pentium III CPU and 256 MB main memory.

4.1 KDSM: KAIST Distributed Shared Memory

KDSM (KAIST Distributed Shared Memory) is a full-blown SVM system, which was implemented from scratch. KDSM was implemented as a user-level library running on Linux 2.2.13. Communication layer is TCP/IP and SIGIO signal handling is used for processing messages from other processes. KDSM uses page-based invalidation protocol, multiple-writer protocol, and support HLRC memory model. Average basic operation costs of KDSM are as follow: 1047μs for fetching a 4KB page, 259μs for acquiring a lock, and 1132μs for barrier (8 processors).

4.2 Applications

We used five applications: TSP, Water, Raytrace_{orig}, Raytrace_{rest} and IS. Raytrace_{orig} and Raytrace_{rest} are two different version of the same program. The former is original application from SPLASH2 [12] and the later is restructured version that removes a lock to protect status information as described in [5]. TSP, Water and IS are from the CVM [7] distribution.

Table 1 shows the problem sizes, number of synchronizations, and their sequential execution times.

Appl.	size	locks	barrs	seq.time
TSP	19 cities	693	2	24.96
Water	343 mol	1040	70	12.96
Raytrace_{orig}	balls4	120945	1	57.82
Raytrace_{rest}	balls4	2081	1	57.82
IS	2^{15},10	80	30	7.05

Table 1. Benchmark applications, problem size, synchronization operations, and sequential executing time

4.3 Experimental Results

Table 2 shows execution results of the benchmarks. It lists 8 processor execution time, speedup, number of page requests, number of pages requests inside critical sections, and message amount. *orig* is base HLRC and *new* is our protocol. It can be seen from table 2 that *new* reduce number of page request and message amount in TSP, Water, Raytrace_{orig}, Raytrace_{new} and consequently obtains performance improvement. Raytace_{orig} uses lock intensively, mostly to protect status information, which is simply 4byte integer variable. *new* eliminates most of the page faults to access this variable, thus reduces 98% of total page requests

Appl.	8-proc.time		speedup		remote getp		getp in CS		Msg.amt(MB)	
	orig	new	orig	new	orig	new	orig	new	orig	new
TSP	7.19	5.83	3.47	4.28	6646	4505	6047	3896	27	19
Water	3.19	3.00	4.06	4.32	2442	2048	852	464	12	10
Raytrace$_{orig}$	182.15	107.88	0.31	0.53	121652	18259	105309	1928	526	129
Raytrace$_{rest}$	10.90	10.71	5.30	5.40	8578	7424	2603	1337	36	31
IS	4.89	4.92	1.44	1.43	4188	4188	2044	2044	25	25

Table 2. Execution results of the Benchmarks

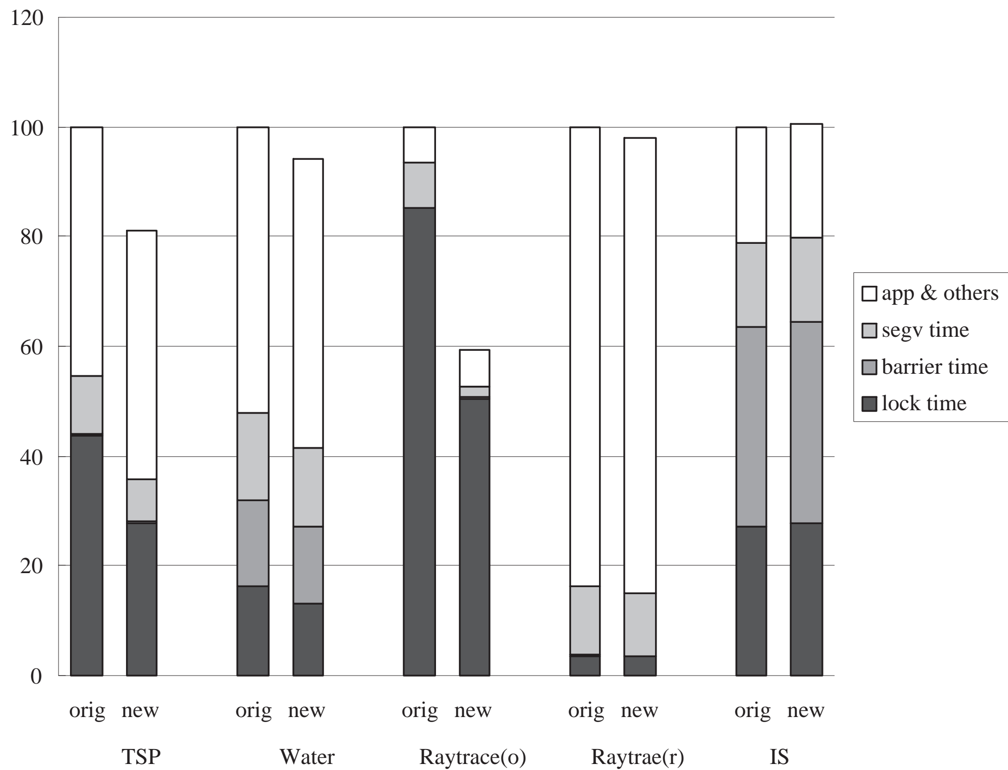

Figure 3. Normalized execution time breakdown

inside critical sections. Moreover, because diffs are transferred instead of pages, message amount is greatly reduced (526MB→129MB). Therefore, Raytrace$_{orig}$ obtains 40% of performance improvement. TSP also shows good performance improvement (19%). Performance improvement is not so good in Water(6%) and Raytrace$_{rest}$(2%). It is mainly because the fraction of page requests inside critical sections is not large compared with total page requests. For IS, it is the only application that doesn't show improvement. In fact performance is slightly degraded (0.6%). Because of large write-granularity inside a critical section diffs are not sent with *new* in IS. The slight time increase is due to the overhead of home page diffing. However, this overhead, as you can see, is very small.

Figure 3 shows normalized execution time breakdown of *new* and *orig*. Page fault handling time (SEGV time), lock time, barrier time, and application time are presented. It can be seen from the figure 3 that reducing page faults inside critical sections is very effective to reduce lock time. For Raytrace$_{orig}$, lock time is significantly reduced because lock is seriously serialized. TSP, Water and Raytrace$_{rest}$ also obtain both time reduction.

5 Related Work

There were many researches to improve lock performance by adapting to migratory access pattern [6, 11, 1]. Although the work described here is done in the context of specific home-based protocol, HLRC, some of our ideas are related with these researches.

Just as our protocol, Lazy Hybrid (LH) [6] protocol selectively updates pages by sending diffs at the time of lock release. However, our protocol differs from LH mainly in that updates are only used for specific pages: accessed pages in the previous critical section of the acquirer. Performance benefits of the updates are limited in the LH because usually more pages are updated than actually needed.

ADSM [11] adapts to the sharing pattern of each page. It dynamically categorizes the type of sharing experienced by each page. For the migratory page (usually protected by lock), ADSM switch the page into single-writer mode and updates that page at the time of lock release. The page selection scheme for the updates is similar with our protocol. However, ADSM transfers a whole page for update because the page is in the single-writer mode.

Amza [1] also adapts to migratory pattern in the similar way but write granularity is considered. If the size of the modification to a page is not larger than some threshold, invalidation based multiple-writer protocol is used to reduce message traffic. However, invalidation protocol suffers from larger delay against update protocol.

6 Conclusions

To improve the lock performance of HLRC, we propose a new lock protocol for HLRC. By updating pages that are expected to be accessed inside a critical section, our protocol reduces page fault handling time inside a critical section, thus avoid lock serialization. The experiments with five applications, which use relatively many locks, shows that our protocol is quite effective. For some applications that lock is the main synchronization operation, our protocol shows significant performance improvement over base HLRC. Overhead of our protocol is negligible in the tested benchmarks.

References

[1] C. Amza, A. Cox, S. Dwarkadas, L. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Procedings of the IEEE*, March 1999.

[2] A. Cox, E. de Lara, Y. Hu, and W. Zwaenepoel. A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory. In *Proceedings of the fifth HPCA*, 1999.

[3] W. H. et al. Home Migration in Home-Based Software DSMs. In *Proceedings of 1st WSDSM*, 1999.

[4] K. Gharachorloo, D. Lenoski, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th ISCA*, 1990.

[5] D. Jiang, H. Shan, and J. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th PPOPP*, 1997.

[6] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, December 1994.

[7] P. Keleher. CVM: The Coherent Virtual Machine. Technical report, 1996.

[8] H. Li, S. Dvarkadas, A. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing '95*, December 1995.

[9] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, September 1986.

[10] W. W. M.C. Ng. Adaptive Schemes for Home-based DSM Systems. In *Proceedings of 1st WSDSM*, 1999.

[11] L. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th HPCA*, February 1998.

[12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th ISCA*, 1995.

[13] Y. Zhou, L. Iftode, and K. Li. Perforamnce Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of USENIX OSDI*, October 1996.