Lock Improvement Technique for Release Consistency in Distributed Shared Memory Systems

Shiwa S. Fu and **Nian-Feng Tzeng**

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Abstract — *Distributed shared memory allows processes to view the physically distributed memory as a globally shared virtual memory. Lazy release consistency (LRC) [5], among known techniques, is an efficient software model proposed for distributed shared memory. It relies heavily on lock synchronization to maintain data coherency. The lock scheme used in LRC, however, conducts many interrupt invocations on the remote processors, which in turn steal effective cpu cycles from remote processors, thus prolonging the lock acquisition time and the total elapsed time of application programs. In this paper, a lock improvement technique is proposed to alleviate interrupt invocations caused by the lock acquire operations, leading to reduction in the lock acquisition time and the overall program execution time. Our improvement technique was evaluated under the TreadMarks' [7] framework using four applications, where TreadMarks is a distributed shared memory system based on LRC. The experimental results indicate that our technique improves the lock acquisition time over TreadMarks on a network of workstations by more than 14% for one application.*

1 Introduction

The shared-memory multiprocessor offers good programmability but often has low scalability. The distributed memory system, on the other hand, exhibits great scalability and usually provides message-passing facilities to allow communication among processors. With the message-passing paradigm, the application program must have full awareness of the system organization, making it relatively difficult to program.

Distributed shared-memory (DSM) systems build the shared-memory paradigm on top of the distributed memory machines, by providing a shared memory abstraction such that the users have a shared-memory environment with message passing taken care of by the DSM layer [12]. As the DSM system has little knowledge about application programs, it may conduct unnecessary message exchanges to maintain data coherency. The issue of how to make fast and efficient data movement is thus important for the design of DSM. Some early researchers in this area resorted to the hardware approach to coherency enforcement [1, 2, 3], while others sought software solutions [4, 5]. The relaxed memory consistency models may incorporate hardware to reduce the latency associated with remote memory accesses [1, 2, 3]. However, the cost involved in special-purpose hardware may pose a major concern. The software approaches, in contrast, put an emphasis on efficient data movement through reducing the number of messages exchanged among processors, because sending messages under such an approach is more expensive than a hardware approach.

In the *release consistency* model [3], shared memory updates do not become visible to a particular processor until certain synchronization access is done (for example, until the *release* of a lock to the next processor). *Eager release consistency* [4] is a software implementation of the release consistency [3], with buffers written till a lock release in order to reduce the number of messages exchanged. The *lazy release consistency* (LRC) model proposed by Keleher *et al.* [5] achieves the best performance among known software approaches. The idea of lazy release consistency is to delay the visibility of data as "lazy" as possible, with the propagation of modifications postponed until the *acquire* time, thus further reducing the number of messages exchanged. All the release consistency models for DSM implementation [3, 4, 5] rely heavily on two primitives for the synchronization access: lock and barrier. In those models, the lock is used to protect the shared data in a mutually exclusive way to avoid *data race*, which leads to an unpredictable result due to multiple contending accesses.

The lock scheme used in the LRC model follows a similar idea as in [8] but was extended to handle *multiple-mutual exclusion*. In LRC, each lock is statically assigned a dedicated manager and any processor (wishing to enter

This material is based upon work supported in part by the NSF under Grants MIP-9201308 and CCR-9300075.

The experimental results of our proposed scheme in this paper are based on a modification of TreadMarks.

1088-4955/96 $5.00 © 1996 IEEE

the critical section) sends a lock request message to the manager of that lock before getting the lock. This request message causes an interrupt at the remote processor in which the manager resides. The operating system at the remote processor then suspends the current executing process and invokes the corresponding interrupt handler. The interrupt handler forwards the request message to the current lock holder, unless the manager itself holds the lock. This forwarded message again causes an interrupt at the remote processor which holds the lock. An interrupt forces the operating system to perform context switch and steals the cpu time from the interrupted process. In addition to the lock request messages, a data request message also invokes an interrupt at the remote processor that holds the required data in the LRC implementation. All these messages tend to interfere with each other at the remote processor if they invoke interrupts at the same remote processor, and the situation gets worse in a multiple-mutual exclusion scenario as more processors are allowed to request for different locks at the same period of time, thus increasing considerably the lock acquisition time and therefore the total application program execution time.

The use of an extra dedicated processor to handle the interrupts to reduce overhead has been investigated and ruled out [6], because it did not pay off to do so, since the spare cycles on a computation processor can be used for handling interrupts. In this paper, we propose a software technique to reduce the number of interrupts caused by the lock request messages, leading to considerable reduction in the lock acquisition time. This is made possible by cooperating the manager and the requesting processor to avoid invoking interrupts whenever possible (as opposed to always forwarding the lock request messages to the remote lock holder). It should be noted that the lock requesting processor, after sending out the request, halts, waiting for the lock to arrive and thus wasting the cpu cycles unproductively (unless the system is multi-threaded). Our scheme takes advantage of the clock cycles, which are otherwise idle, in the requesting processor and replaces forward messages with reply messages to save the number of interrupts, whenever possible. Four benchmark programs: Traveling Salesman Problem (TSP), Integer Sort (IS), Water, and Quicksort (QS) are conducted to evaluate the performance under the TreadMarks' framework. When compared with original TreadMarks implementation, our proposed technique saves the average lock acquisition time by up to 14.7% for TSP, 7.9% for IS, 2.0% for Water, and 1.2% for QS, on a network of eight workstations.

2 Previous Approach

As our work was done under the TreadMarks' framework, we first describe TreadMarks and its lock scheme in the following. Our lock technique and its details are stated in Section 3.

2.1 TreadMarks and Its Lock Scheme

TreadMarks [7] is a distributed shared memory system developed at Rice University. It is implemented entirely as a user-level library on top of Unix and is a multiple-writer protocol based on lazy release consistency (LRC) [5], which allows a page to be modified by several processes at a time. The notification of those modifications is achieved by piggybacking the write notice into the lock grant message, where a write notice is an indication that a page has been modified in a particular time interval. When a process later accesses the page, all modifications to the page are requested, according to information in the received write notice(s), so that data coherency in the page can be enforced.

TreadMarks employes a token-based lock scheme to handle multiple-mutual exclusion. Every lock is handled by a fixed manager (i.e., a processor), and processors are assigned as lock managers statically, as follows: processor P_X is the manger of lock I in a system with N processors, if I mod N equals X, with $0 \leq X \leq (N\text{-}1)$. Two types of messages are used to establish the order for all lock request processors to enter the critical section in sequence, i.e., REQT and LOCK messages, which stand for requesting lock messages and lock passage messages, respectively. Any processor with the lock is allowed to enter the critical section immediately. A processor, say P_X, requesting lock(I) always sends a request message, which contains its processor ID (denoted by REQT(X)), directly to the manager of that lock. The manager makes it possible to establish a distributed waiting queue (of all processors requesting for lock(I)) by recording (1) the processor which has most recently requested the lock (designated as the tail processor in the distributed waiting queue) and (2) the processor which is the next to enter the critical section. This is accomplished by employing two local variables "lock(I).tail" and "lock(I).next", respectively. The manager may or may not be the lock holder, when it receives an REQT(X). If it is not the lock holder nor is requesting the lock, the manager, on receiving an REQT(X) message, forwards REQT(X) to the tail processor (of the queue) and updates its record by setting lock(I).tail's value to X. On the other hand, if the manager has requested the lock or is in use of the lock at the time when it receives an REQT(X) message, it simply adds the request to its waiting queue (by setting lock(I).tail and lock(I).next to X) without forwarding the message, as it already issued one for itself earlier to the lock holder. If any further message, say REQT(Y), arrives at the manager after REQT(X), the manager forwards REQT(Y) to the tail processor X and updates its record by setting lock(I).tail's value to Y, be-

cause processor Y at this moment is the next processor of the current tail X and is the new tail of the waiting queue. On receiving an REQT(X), the lock holder, if not in need of the lock, sends the privilege to the requestor (X) directly using a LOCK message. An attractive property of this algorithm is that it always takes three (3) exchange messages for a requestor to get the lock, if the manager does not own the lock, and it takes only two (2) messages if the manager holds the lock.

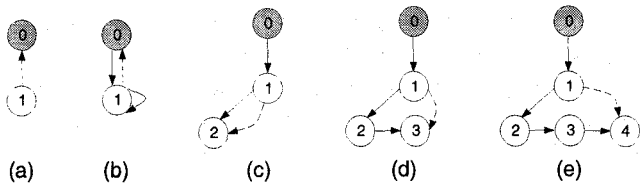

Fig. 1. Lock acquisition in TreadMarks.

The above idea is better illustrated by an example. Consider a system with 8 processors and the situation where the access privilege of lock *1* is at processor P_0 initially, as depicted by gray circles in Fig. 1. The local variables lock(*1*).next and lock(*1*).tail at each processor are illustrated by the solid arrows (i.e., pointers) and the dashed arrows (i.e., pointers), respectively, in Fig. 1. Now, five processors (P_0, P_1, P_2, P_3, and P_4) wish to request lock(*1*) in sequence. Since P_0 possesses lock(*1*), it enters the critical section immediately. Assume that P_1 issues its request to the critical section before P_0 finishes with the critical section, and the manager of lock(*1*) is P_1. Since the manager, P_1, knows the lock holder, it issues an REQT message to P_0 (the current lock holder), requesting the lock access privilege. After sending out the request message, REQT(*1*), P_1 updates its local variable lock(*1*).next, pointing to itself. On receiving REQT(*1*), P_0 sets its variable lock(*1*).next to P_1 (i.e., pointing to P_1), as shown in Fig. 1(b). Next, P_2, P_3, and P_4 issue REQT(*2*), REQT(*3*), and REQT(*4*), respectively, to the lock manager, P_1, in order to get the lock access privilege. On receiving the REQT(*2*) message from P_2, P_1 simply adds the request to its waiting queue (without forwarding it to the lock holder) by setting its lock(*1*).next and lock(*1*).tail to P_2, as depicted in Fig. 1(c). When the request, REQT(*3*), from P_3 arrives at P_1, it is forwarded to the tail processor P_2, with lock(*1*).tail at P_1 set to P_3, as demonstrated in Fig. 1(d). P_2 updates its lock(*1*).next to P_3 on receiving the forwarded REQT(*3*) message (from P_1). Similarly, when the REQT(*4*) message from P_4 reaches P_1, it is forwarded to the tail processor P_3, which will update its lock(*1*).next on the receipt of the forwarded REQT(*4*) message to form the waiting queue, as shown in Fig. 1(e).

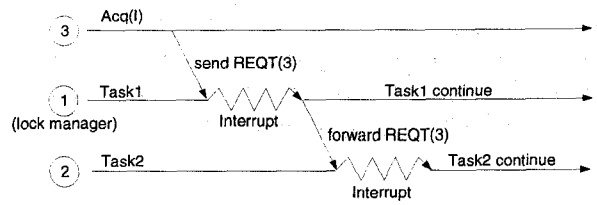

Fig. 2. Detailed interrupt invocations caused by a request message in TreadMarks.

The implementation of the above scenario in TreadMarks involves software interrupt invocations; specifically, each request message (or the forwarded request message) incurs one interrupt at the remote processor. The interrupts invoked by the request message issued by P_3 (corresponding to the transition from Fig. 1(c) to Fig. 1(d)) are detailed in Fig. 2. The interrupts invoked by REQT(*3*) and the forwarded REQT(*3*) message steal cpu cycles from their respective Task1 and Task2 that were interrupted. The above scenario becomes more complicated when multiple locks are involved. Consider the situation where P_2 and P_3 (in the waiting queue) in Fig. 1 are the managers of lock(*2*) and lock(*3*), respectively. If any other processors request these locks during the period of lock(*1*) invocations, those requests tend to stall each other at processors P_1, P_2, and P_3. Furthermore, any processor might receive data requests from other processors, and those requests might interfere with each other. Finally, a processor which is waiting for the lock privilege, on receiving an interrupt, pays the cpu time to do context switches and to execute the interrupt handler, thus delaying lock passage. As a result, the interrupts caused by those messages extend the lock acquisition time, prolonging the elapsed time of application programs. In light of this phenomenon, we propose a technique to reduce the number of interrupt invocations caused by the lock request messages, as described in the next section.

3 Proposed Approach

A new type of message, REPLY(X), is employed in the proposed technique, where X is the processor ID (pointed by manager's lock(*1*).next) piggybacked into the REPLY message. This message is sent by the manger to the requesting processor to form the waiting queue.

3.1 Basic Idea

TreadMarks [7] is a distributed shared memory system based on lazy release consistency (LRC) [5]. The reason for TreadMarks to incorporate the interrupt handler (i.e., SIGIO signal) is to minimize latency in handling incoming requests. However, we found that not all messages

were beneficial from using interrupts. At least, interrupts caused by the lock request message described in the last section prolong considerably the lock acquisition time. In fact, these interrupts can be avoided by cooperating neatly the manager and the requesting processor, as stated next.

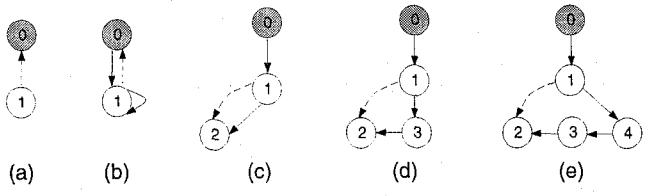

Fig. 3. Lock acquisition in the proposed technique.

A processor, after sending out a request, is in the state of waiting for the lock to come, and an arrived reply message or lock message causes no interrupt. The basic idea of our technique is that the manager, on receiving a request message, sends a reply message, which includes lock(I).next value (i.e., REPLY(lock(I).next)), to the requestor (instead of forwarding the message to the tail processor in the waiting queue, as in TreadMarks), and then sets lock(I).next value to the requestor's processor ID (i.e., the requestor becomes the next element in the waiting queue). On receiving a REPLY(Y) message from the manager, the requestor sets its lock(I).next to Y. Noted that the reply message causes no interrupt at the requestor, which is in the waiting state. However, forwarding the message to the tail processor in the waiting queue causes an interrupt at the tail processor. Consider the transition from Fig. 1(c) to Fig. 1(d) as an example. On receiving the request message from P_3, manager P_1 issues a REPLY(2) message to requestor P_3 and updates its lock(I).next to be P_3; P_3 updates its lock(I).next to be P_2 after receiving the REPLY(2) message, as shown in Fig. 3(d). A similar process happens to P_4, as illustrated in Fig. 3(e). Comparing Fig. 3 and Fig. 1, we find that pointers in those waiting queues are in the opposite directions.

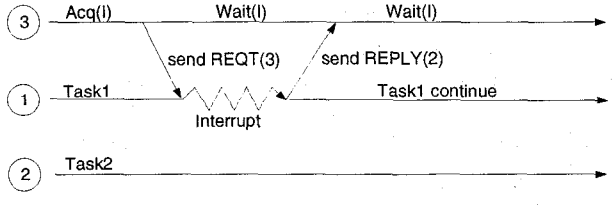

Fig. 4. The details of interrupt savings due to the proposed technique.

The detailed interrupt invocation of the above instance from Fig. 3(c) to Fig. 3(d) is depicted in Fig. 4, where the REPLY(2) message invokes no interrupt at the requestor P_3, unlike that in Fig. 2.

3.2 The Algorithm

The proposed technique behaves in the same way as the lock scheme in TreadMarks, if the manager is not in request for the lock (i.e., not in the queue) nor is in use of the lock. On the other hand, if the manager is not in either one of the two cases, a lock waiting queue (for example, the queue which contains P_3, P_4 in Fig. 3(e)) is formed through the REPLY message. Although this waiting queue is in the LCFS (i.e., last come first served) order, no starvation would occur, because once the manager finishes with the the critical section and passes the token to its 'next' processor, any lock request which arrives from then on, will be added to the waiting queue in the first-come-first-served manner (since the manager then behaves in the same way as the lock scheme in TreadMarks). The number of interrupt invocations caused by our lock scheme is lowered, thus reducing the lock acquisition time. Since one REPLY message is produced in place of one forwarded REQT message, the number of total messages generated under our proposed technique is about the same as that in TreadMarks. As the REPLY message is issued by the manager to form a lock waiting queue, the total number of REPLY messages can serve as a gauge for potential performance improvement due to our proposed scheme, as discussed in Section 5.

The proposed lock technique consists of three components: Lock_acquire(I), Lock_release(I), and Req_handler(I, REQT(Y)), as demonstrated by C-like codes in Fig. 5, Fig. 6, and Fig. 7, respectively. These components are different from those found in TreadMarks, implemented to accommodate our proposed lock scheme. The first two components provide the interfaces for the application program to enforce and release the lock(s) on mutually exclusive shared data, while the last component is invoked by an interrupt signal (for example, the SIGIO signal in Unix) whenever a processor receives a request message. Each processor in the proposed lock technique uses an array variable *lock* to record the status of all locks. This variable contains three fields for each lock: **next**, **tail**, and **held**, which keeps the next processor, the tail processor, and whether the current processor is in request for or in use of that lock, respectively. Initially, both **next** and **tail** are set to 0 (i.e., processor 0 holds all the locks) and **held** is set to false.

If an application program requests for lock I, it calls Lock_acquire(I) to get the lock. If the current processor X does not have the lock, a request message is issued to the manager (or to the tail, if X is the manager); otherwise, the application program gets the lock privilege immediately.

```
Lock_acquire(I) {      /* processor X requests lock I */
    lock(I).held = TRUE;    /* set requesting flag */
    if (lock(I).next != X) {   /* X does not have the lock */
        lock(I).next = X;
        send REQT(X) to manager,
            or to lock(I).tail if X is the manager of lock I
        wait until receiving a REPLY or a LOCK message;
        if (receive a REPLY(Y) message) {
            lock(I).next = Y;
            wait until receiving a LOCK message;
        }
    }
}
```

Fig. 5. Lock acquire.

After the application program finishes with the critical section, it calls Lock_release(*I*) to release lock *I*. The lock is then consigned to the next processor, if any; otherwise, the lock privilege stays there until it is requested later on. In addition, the **held** field is set to false (i.e., not in request for nor in use of the lock).

```
Lock_release(I) {      /* processor X releases lock I */
    lock(I).held = FALSE;    /* reset requesting flag */
    if (lock(I).next != X)   /* there has been a request */
        send LOCK message to lock(I).next;
}
```

Fig. 6. Lock release.

Req_handler() is invoked when a processor receives a request message.

```
Req_handler(I, REQT(Y)) {  /* an REQT(Y) for lock I */
    if (lock(I).next == X) {  /* X itself is tail in queue */
        if (lock(I).held != TRUE)
            send LOCK to processor Y;
        lock(I).next = Y;
        if (processor X is the manager of lock I)
            lock(I).tail = Y;
    }
    else if (processor X is the manager of lock I) {
        if (lock(I).held != TRUE) {
            forward REQT(Y) to processor lock(I).tail;
            lock(I).tail = Y;
        }
        else {
            send REPLY(lock(I).next) to processor Y;
            lock(I).next = Y;
        }
    }
}
```

Fig. 7. Lock request handler.

4 Benchmark Programs

Four application programs are employed to evaluate the performance of our proposed technique: Water, TSP, IS, and QS. As the proposed technique could save only the lock management time, no application without locks is selected. In fact, we ran Barnes (a benchmark program without any lock involved from [10]) on both TreadMarks and TreadMarks with our technique incorporated, and found that they delivered the same performance. TSP is the one that uses lock primitives only, while all the other three involve both barrier primitives and lock primitives for synchronization. Water and QS are originally from the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite [10], IS comes from the NAS benchmark suite [11], and TSP was developed at Rice University [9].

4.1 Water

Water is an N-body molecular dynamics simulation that evaluates forces and potentials in a system of water molecules in the liquid state. The molecules are distributed equally among processors, with the computation performs over a number of user-specified time steps. Each time step contains two phases: force computation phase and displacement computation phase, separated by barrier synchronization. A lock protects access to each molecule record during the force computation phase as each force value is updated by several processors, while no lock is required during the displacement computation phase because each processor only updates the displacements of its own molecules

A slightly modified version of Water provided by Rice University [9] is used in this study, where each processor employes a local variable to accumulate its updates to a molecule's record during an iteration, and the accumulated updates are then applied to the shared record (protected by a lock) at the end of each iteration. Thus, the modified version greatly reduces synchronization overhead by condensing multiple lock acquisitions into single acquisitions. It simulates 343 molecules for 5 time steps in this study.

4.2 Traveling Salesman Problem (TSP)

TSP is to find the shortest path (a *Hamiltonian circuit*) among all cities in a given problem. The Hamiltonian

circuit starts from a designated city, passes through every city exactly once, and returns back to the start city. This complete path is termed as a *tour* in [9], and it is assumed that all cities are fully connected with an associated weight between any two cities.

As TSP is an *NP-complete* problem, the program in [9] uses a branch-and-bound algorithm to solve it. The program has a shared global queue (protected by a lock) of partial tours. Each processor gets a partial tour from the queue, extends the tour, and returns the results back to the queue. The input size of 18 cities is considered in this study.

4.3 Integer Sort (IS)

Integer Sort ranks an unsorted sequence of N keys. The *rank* of a key in a sequence is the index value i that the key would have if the sequence of keys were sorted. All the keys are integers in the range $[0, B_{max}]$. A bucket sort is used to do the ranking.

Each processor first ranks its set of keys. It then requests an exclusive access to a shared array, increases the values in the shared array with its own rankings, and stores a copy of the current values in the shared array. Processors are synchronized through barriers between ranking, and through locks during ranking. In this study, the values of N and B_{max} are 2^{20} and 2^9, respectively.

4.4 Quick Sort (QS)

QS, a recursive sorting algorithm, uses a centralized task-queue based approach for sorting an array of integers, where a processor repeatedly dequeues an unsorted sub-array from the task queue and recursively applies the quicksort algorithm to the dequeued unsorted element. The entire array is inserted in the task queue initially. According to the quicksort, a dequeued unsorted element is partitioned into two sub-array around the pivot, and the processor keeps working on the larger portion of the partition while the smaller part is enqueued in the task queue. The base case of the recursion is encountered when the partition size reaches a threshold of 512 integers, at which time the partition is sorted locally by a bubblesort algorithm.

An array of 100K integers is considered in this study.

5 Performance Evaluation

5.1 Platform and Implementation

Our simulation program is based on the TreadMarks' framework, with only the lock scheme replaced by our proposed technique and all the remaining TreadMarks codes kept unchanged. The experimental work was conducted on eight (8) Sun 4/75 SPARCstations running SunOS4.1.4. All Sun workstations are RISC processors with the speed of 40 MHz and are connected by an Ethernet, operating at 10Mb/sec. Each processor has its own local memory, and communication among processors is achieved by using UDP/IP as the message transport protocol.

TreadMarks implements intermachine communication by using the Berkeley sockets interface and employs a SIGIO signal (interrupt) handler to handle incoming requests. Message arrival at any socket which is for receiving request messages (i.e., REQT messages) generates a SIGIO signal, while any socket which is for receiving the response messages (i.e., REPLY and LOCK messages) is designed to produce no interrupt signal (because the receiver of response messages is waiting for a message to arrive). The proposed lock technique makes good use of the socket for response messages in order to reduce the number of interrupts.

Table 1. Results of the proposed technique and TreadMarks

Program	Algorithms	Elapsed time (secs)	Lock time (μs)	Total msgs	SIGIO msgs	Total lock requests	REQT msgs	Total data (Kbytes)	REPLY msgs
Water	Ours	28.59	3663	32338	19221	8742	8178	11377	62
	TreadMarks	29.26	3736	32228	19213	8742	8139	11373	0
TSP	Ours	19.30	42869	7892	4074	462	436	1954	40
	TreadMarks	19.75	50276	7968	4142	461	437	1982	0
IS	Ours	2.34	53327	595	287	41	38	666	28
	TreadMarks	2.39	57876	601	319	41	40	666	0
QS	Ours	15.54	63680	11016	5368	889	883	11984	496
	TreadMarks	15.94	64449	11237	5978	900	894	12035	0

5.2 Results

Results of the proposed technique and TreadMarks are provided in Table 1, where they are averaged over 30 independent runs for each application program. Seven parameters of interest are listed in this table: total elapsed time of the application program, lock acquisition time, total number of messages transferred over the network, the number of messages that invoke the SIGIO signal (called *SIGIO message*), the number of lock requests, the number of REQT messages, the total amount of data movement over the network, and the number of REPLY messages produced. The lock acquisition time is the ratio of total time in acquiring the lock to the number of lock requests. It is found from Table 1 that the proposed technique outperforms TreadMarks with respect to most parameters for all the four application programs (note that TreadMarks produces no REPLY message).

Table 2. Performance improvement of the proposed technique

Program	RR rate	Lock types	Improvement in lock acquisition time
Water	0.76%	lock(0)~lock(7)	2.0%
TSP	9.17%	lock(1), lock(2)	14.7%
IS	73.68%	lock(1)	7.9%
QS	56.17%	lock(1), lock(2)	1.2%

The use of REPLY messages in our technique leads to fewer interrupt invocations at the remote processors. We define *RR rate* as the ratio of the number of REPLY messages to the number of REQT messages. The RR rate can be regarded as an indicator to gauge the degree of possible performance improvement due to the proposed technique. The RR rates of the application programs are provided in Table 2. The lock acquisition time improvement percentage of our technique over TreadMarks is also listed in Table 2. From Table 2, IS has the best RR rate among all, and yet it does not have the highest improvement percentage in lock acquisition time. Instead, TSP enjoys a better improvement percentage. Two factors decide the lock acquisition time (also the total elapsed time to be discussed later): (1) the number of locks and (2) the number of messages that invoke the SIGIO signal (known as SIGIO messages). If a program involves more locks, the processors in the waiting queue are more likely to be mangers of other locks, and thus there could be more interrupt invocations to stall each other at these processors. In addition to the lock request messages, data request messages may cause interrupt invocations at remote processors as well, aggravating the degree of interrupt interference.

```
Lock_acquire(1);
for (i=0; i < MAXKEY; i++)
    access shared data;
Lock_release(1);
```

Fig. 8. Lock in IS.

IS involves only one lock throughout its lifetime (see Table 2), while TSP contains two locks. IS uses lock(*1*) to protect the critical section of a loop, as illustrated in Fig. 8. A processor executing this code is likely to be in the state of waiting for the lock to arrive. Multiple processors may be at the waiting state at a given point of time, and interrupt invocations at those waiting processors will not stall any productive process, thereby causing a lighter adverse impact on the IS performance. In contrast, there are two locks in TSP, and in particular, the accesses to these two locks are done in an interleaved way for TSP. As a result, TSP outperforms the other three, although its RR rate is only 9.17% (compared with 73.68% for IS). Similarly, Water, utilizing 8 locks on a system with 8 workstations, achieves an improvement of 2.0%, despite its low RR rate of 0.76%. Finally, the improvement percentage of QS is the smallest, though it has two locks and a high RR rate. This is mainly because it works on one lock which has a structure similar to that in IS as shown in Fig. 8, where multiple processors may be at the waiting state at a given point of time. The average lock acquisition time for each program is illustrated in Fig. 9.

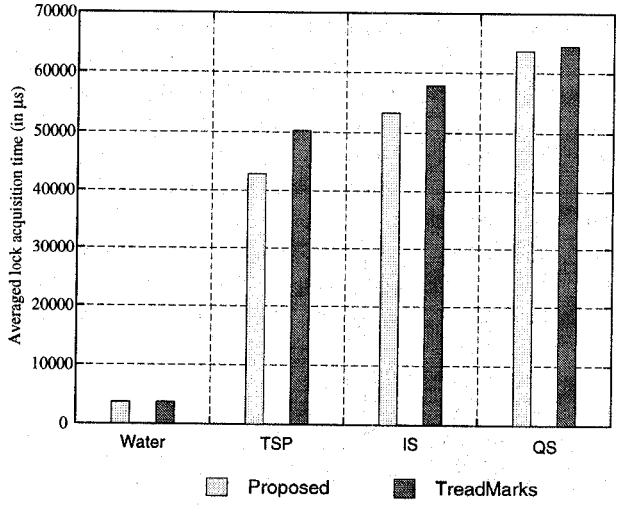

Fig. 9. Lock acquisition time.

The total elapsed time for each program is depicted in Fig. 10, where the proposed technique outperforms TreadMarks consistently. Although Water has only 0.76% RR rate and IS has 73.68% RR rate (see Table 2), the former enjoys a larger elapsed time improvement than the later. This is because Water has 8 locks and 19K SIGIO messages (see Table 1), in contrast to one lock and about 300 SIGIO messages for IS. This large amount of interrupt invocations tends to stall each other, especially under multiple locks. Both TSP and QS contain two locks each and achieves a similar elapsed time improvement.

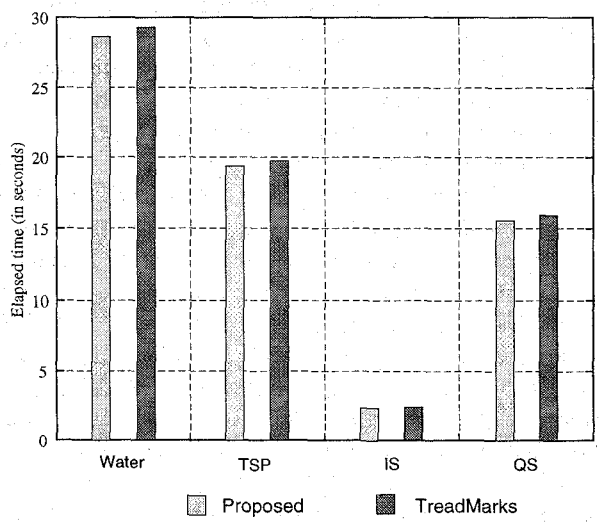

Fig. 10. Total elapsed time.

6 Conclusions and Future Work

In this paper, a token-based software lock improvement technique for the distributed shared memory system established using TreadMarks has been evaluated. Our experimental environment is based on the TreadMarks' framework, with the lock scheme replaced with the proposed technique and all the other TreadMarks' codes kept unchanged. It is observed from our study with four application programs that the manager of a lock in our technique is likely to issue a REPLY message instead of forwarding a request message, consequently reducing the number of interrupt invocations at remote processors and therefore lowering the lock acquisition time. While the queue of waiting processors is in the LCFS order when the manager is in request for or in use of the lock, as a result of the proposed lock technique, no processor could wait for the lock indefinitely, free from starvation. The experimental results demonstrate that the proposed technique gives rise to improved performance under all the four application programs, in terms of total elapsed time and the mean lock acquisition time.

Three factors may affect the degree of performance improvement due to the proposed technique: the RR rate, the number of locks, and the number of SIGIO messages. In general, the larger in any of the three values, the more pronounced performance improvement. The number of locks and the number of SIGIO messages have particularly noticeable impacts on performance. We shall examine the performance of other applications with more locks in the future. Also, a larger system size is expected to result in higher performance improvement due to the proposed technique, because the three factors mentioned above tend to increase as more processors are involved. The performance of our proposed technique behavior under a larger system, such as the IBM SP/2 machine, is under investigation.

References

[1] S. Adve and M. Hill, "Weak Ordering: A New Definition," *Proc. 17th Int. Symp. Computer Architecture,* pp. 2–14, May 1990.

[2] M. Dubois and C. Scheurich, "Memory Access Dependencies in Shared-Memory Multiprocessors," *IEEE Trans. Software Engineering,* Vol. 16, No. 6, pp. 660-673, June 1990.

[3] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Int. Symp. Computer Architecture,* pp. 15–26, May 1990.

[4] J. B. Carter *et al.*, "Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems," *ACM Trans. Computer Systems,* Vol. 13, No. 3, pp. 205–243, August 1995.

[5] P. Keleher *et al.*, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Int. Symp. Computer Architecture,* pp. 13–21, May 1992.

[6] M. Karlsson and P. Stenstrom, "Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers," *Proc. 2th Int. Symp. High Performance Computer Architecture,* pp. 4–13, February 1996.

[7] Cristiana Amza *et al.*, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computers,* pp. 18–28, February 1996.

[8] M. L. Neilsen and M. Mizuno, "A Dag-Based Algorithm for Distributed Mutual Exclusion," *Proc. 11th Int. Conf. Distributed Computing Systems,* pp. 354–360, May 1991.

[9] P. Keleher, *Distributed Shared Memory Using Lazy Release Consistency,* Ph.D thesis, Rice University, December 1994.

[10] J. P. Singh *et al.*, "SPLASH: Stanford Parallel Applications for Shared-Memory," Tech. Rep. CSL-TR-91–469, Stanford University, April 1991.

[11] D. Bailey *et al.*, "The NAS Parallel Benchmarks," Tech. Rep. TR RNR-91–002, NASA Ames, August 1991.

[12] N.-F. Tzeng and P.-C. Yew, "Special Issue on Distributed Shared Memory Systems," *Journal of Parallel and Distributed Computing,* Vol. 29, No., 2, September 1995.