Aggressive Release Consistency for Software Distributed Shared Memory

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

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

Abstract

As a software-based distributed shared memory (DSM) system is especially sensitive to the traffic amount over the network, we propose in this paper a new software DSM model which postpones the enforcement of data coherence at the time of the first shared memory access after an acquire, instead of at the time of the acquire like the lazy release consistency (LRC) model, leading to an aggressive implementation of release consistency and thus a reduced number of messages transferred over the network when compared with LRC. Our model is evaluated on the basis of the TreadMarks' [7] framework using three applications, where TreadMarks is a software DSM implementation following LRC. The experimental results on a network of workstations indicate that our model leads to fewer messages transmitted across the network than LRC, by over 16% for one application and over 12% for the other two.

1 Introduction

Programming for the distributed memory system has to take care of data movement and partition, which is tedious and normally machine-dependent. Parallel codes developed for the distributed memory system are likely to be of poor programmability and portability. In contrast, a shared memory system has a global memory address space shared by all processors; it offers an intuitive programming paradigm which allows data sharing and coordination through the common memory address space. Such a system naturally leads to high programmability and portability of parallel codes, saving software development and deployment costs. It is possible to construct a layer of shared memory abstraction on top of the distributed memory system so that programmers can envision a globally shared address space and develop codes in a similar way as for shared memory systems. This distributed shared memory (DSM) system aims to remove the problems associated with the distributed memory system. The code developed for a DSM system, like that for a shared memory machine, is often more comprehensible because it involves no remote communication instructions nor machine-dependent data partition. Existing software for shared memory machines can be easily modified for execution on DSM systems to preserve software investment. Due to its potential advantages, DSM has been an active research area, with many prototype systems implemented and demonstrated [2, 3].

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 model in this paper are based on a modification of TreadMarks.

The address space of a DSM system is distributed across memories at interconnected processors. To reduce traffic over the network, a replication of certain data stored at a remote processor is usually created in the local cache of a processor. Multiple copies of data, while improving read performance, pose the need of coherence enforcement, which can be achieved through hardware, software, or a combination of the two [2]. Apparently, the software-based implementation is most attractive to the majority of existing networks of workstations (NOWs) [1], since it involves no added hardware, which tends to be machine-dependent and expensive. For a software-based DSM system, coherence traffic overhead is to be minimized, because such a system is especially sensitive to the traffic amount over the network. To this end, the release consistency model [4] seems to be a preferred choice for the software-based DSM implementation due to its low amount of coherence traffic. Release consistency requires that synchronization operations for any access to shared variables employ the lock *acquire* instruction, which forces coherence of shared data observed by the processor issuing the acquire instruction. It does not guarantee that shared memory is consistent all of the time, but rather making sure consistence only after synchronization operations. This naturally results in lower communication traffic due to coherence than a more restrictive memory consistency model, such as sequential consistency.

Previous software-based DSM [5, 6, 12] systems have attempted to reduce the number of messages and the amount of data transferred among processors by various degrees. In particular, an *eager* software implementation of release consistency, found in Munin [11], buffers propagating the updates of shared data at each processor until the execution of a *release*, so that all writes going to the same destination are merged into a single message. Later on, *lazy* release consistency (LRC) was considered [6], in an attempt to further postpone the propagation of updates until the *acquire* (to the shared data made by another processor) following a release, leading to more reduction in the number (and amount) of messages (and data) exchanged. Instead of sending updated shared data to all other processors at a release as in eager release consistency [5], this LRC (1) enables indications of updated shared data (instead of the data itself) to be piggybacked in the lock release message sent to the processor which acquires the shared data, in order to reduce the number of messages (and data) involved for coherency enforcement, and subsequently (2) avoids sending messages unnecessarily to those processors which do not require the shared data; in other words, the updated shared data is sent only on the demand of the processor which acquires the shared data. It has been demonstrated that LRC generally outperforms eager release consistency, with respect to the number of

0-8186-7813-5/97 $10.00

messages and the amount of data exchanged [6].

In this paper, we propose a new software-based DSM model, called *aggressive release consistency* (ARC for short), to further reduce the amount of traffic over the network, where the propagation of modifications is "aggressively" postponed until the *first shared memory access* after the lock acquire (instruction). It should be noted that every processor in LRC, before fetching shared data, must first issue a lock request message to acquire the lock and then sends a data request message (according to the received indications) to get the desired updated shared data, where the lock request message and the data request message are issued in order and separately. Any processor, on receiving a lock request (or data request) message, releases the lock (or sends the requested data) to the requestor in two separate messages too. Totally, up to four messages are required to satisfy the access of one shared data after the lock acquire instruction. In contrast, our proposed ARC model enforces data coherence at the time when a processor accesses the first shared data after the lock acquire instruction, making it possible to merge a lock request message and a data request message into one message; correspondingly, a lock release message and a data reply message can be merged into one message too. As a result, ARC takes only two messages, rather than four messages like LRC, for each coherence enforcement, leading to reduction in the number of lock messages and data messages by as much as 50% in the best scenario. We have implemented our proposed ARC on the basis of the TreadMarks' [7] framework for evaluating its performance, where TreadMarks is a software-based DSM implementation for LRC. Our implementation is experimented on a Ethernet-connected NOW of size 8 using three benchmark programs: QuickSort (QS), Integer Sort (IS), and Traveling Salesman Problem (TSP). When compared with LRC, ARC saves the total number of messages transmitted by up to 16.5% for QS, 12.3% for IS, and 12.7% for TSP.

2 Aggressive Release Consistency

The basic idea of ARC is introduced in Section 2.1, followed by its properties and potential in Section 2.2 and Section 2.3, respectively. Our ARC implementation is presented in Section 3.

2.1 Basic Idea

As software-based DSM is especially sensitive to the traffic amount over the network, both Munin and LRC enforce data coherence at the page level instead of at the cache-line level common for hardware-based DSM [4]. To overcome the coarse granularity arisen from the enforcement of data coherence in the page level, software-based DSM usually employs a *multiple-writer* protocol, which allows different parts of the page to be modified by several processors concurrently such that false sharing can be alleviated without generating an unnecessarily large amount of coherency traffic. In LRC, the propagation of modifications is postponed until the time of the acquire. The notification of those modifications is achieved by piggybacking the *write notices* into the lock release message, where a write notice is an indication that a page has been modified in a particular time interval, rather than the actual modification itself; the use of write notices could reduce the data amount transmitted and also avoid unnecessary data propagation. When a processor later fetches a page, all modifications to the page in a particular period of time are requested, according to information kept in the received write notice(s), so that data coherency in the page can then be enforced.

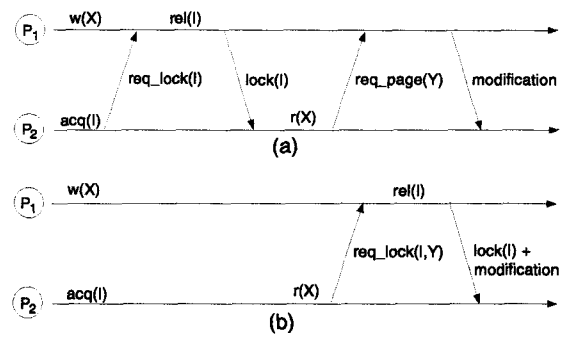

Fig. 1. Messages in response to a shared memory access under (a) LRC, and (b) ARC.

Consider an example shown in Fig. 1(a) with the following events: (1) processor P_1 writes data to memory location X which resides in page Y, (2) processor P_2 attempts to get lock l by sending a lock request message, req_lock(l), to the lock holder P_1, (3) P_1 replies a lock release message, lock(l), to P_2, with its write notice piggybacked in the message, (4) P_2 reads data miss at location X and then sends a data request message, req_page(Y), to P_1 so as to get updated data in page Y (according to the received write notice), and (5) finally P_1 responds with a data reply message (with the modification). Totally, four messages are required for this acquire-release processor pair in the example.

In our proposed ARC model, however, P_2 in the above example holds its lock request message until a subsequent shared memory access occurs (a read miss at location X in this case), and then sends both messages (lock request and data request) together using a combined message, req_lock(l, Y), to P_1, as depicted in Fig. 1(b). Later on, P_1 replies a lock release message, in which the write notice plus the modification is piggybacked, to P_2. In this way, only two messages are required (instead of four as in LRC). It should be noted that a data request message normally contains a message header plus the requested page number (which takes only one or two bytes) and other information, and piggybacking the page number in the lock request message actually reduces both the total number of messages and the amount of data transferred over the network. While reduction in the total amount of data transferred lowers traffic overhead due to coherency enforcement, the most pronounced advantage resulting from ARC lies in its reduced number of messages.

2.2 ARC Properties

According to release consistency [4], all shared memory accesses that precede the acquire must be performed at the acquiring processor before this processor may continue past an acquire. Eager release consistency [5] enforces this property by propagating modifications envisioned by the releasing processor to all the other processors at the time

of release, while LRC [6] guarantees this property by propagating write notices (instead of the modified data itself) via the lock release message that effects a release-acquire pair, so as to reduce the amount of transmitted messages and data. Under our ARC model, the enforcement of data coherence is further postponed until the *first shared data access* after the lock acquire, rather than immediately at the *acquire*, like LRC. Since memory accesses, if any, between the acquire instruction and the first shared data access instruction involve only local variables, the execution of these local memory accesses never compromises the integrity of shared memory consistency. As a result, ARC guarantees the same data consistency as LRC, which has been shown to exhibit identical data consistency as release consistency [4]. This postponement in enforcing data consistency due to ARC yields effective reduction in the number of messages and the amount of data transferred, as will be illustrated by experimental results in Section 5.

Like any previous software-based DSM, ARC enforces data consistency in the page level and employs a multiple-writer protocol with the propagation of modifications (instead of the whole updated pages) to overcome the coarse granularity and false sharing commonly found in software-based DSM. The propagation of modifications in ARC adopts the same strategy used by LRC in order to save the traffic amount transferred over the network. Specifically, write notices are piggybacked in lock release messages and the data reply messages (with modifications) are transferred only on the demand.

According to our ARC model, any access miss toward the shared location brings the required modifications from all the processors which updated the shared location to be fetched. It is thus highly possible that a processor at the time of releasing the lock has accumulated most of the modifications (for the page of interest) resulting from the updates of other processors and of the processor currently executing the critical section. This *modifications accumulation* property is advantageous to our ARC model (and also the LRC model), because the acquiring processor under ARC can get most modifications for the requested page from the current lock holder using the lock request message (piggybacked with the requested page number), without issuing data request messages separately to other processors which updated the requested page previously.

2.3 Further Potential

ARC presented here combines only one lock acquire with a subsequent shared memory access. In general, an application program may employ multiple lock acquires, as shown in the example. In this case, ARC may enforce data coherence at the first shared memory access (in page *Y*) after both lock acquires, rather than at the second lock acquire (to lock *J* in the example), amid the expectation for merging the two lock request messages (with respect to lock *I* and lock *J*) and the data request message (with respect to the fetched page *Y*) into a single message. This mergence of three messages into one is achievable, as long as the two lock request messages head toward the same destination processor. If the two lock messages have different destinations, they are issued separately, with each one piggybacking the fetched page number, *Y*. Thus, ARC always leads to reduction in the number of messages, by 2 (or 1) in the former (or latter) situation for each occurrence.

Example: two lock acquires below.
Lock_acquire(*I*);
local memory accesses;
Lock_acquire(*J*);
first shared memory access toward page *Y*;

The above example considers data coherence in connection with only lock primitives. If barrier primitives are also taken into account, more combinations among locks, barriers, and data request messages are possible, and consequently a larger savings in the number of messages (due to message mergences) is likely to result. In addition, investigation into the best number of instructions between a lock acquire (or barrier) and the first shared memory access instruction after the acquire (or barrier) could unveil useful information for instruction scheduling to optimize system performance, because the number of instructions thereinto dictates how late the lock request (or barrier) message will be issued under ARC. Earlier issuance of a lock request (or barrier) leads to the lock to arrive sooner, but has the possible disadvantage of generating more data reply messages and more transferred data.

3 ARC Implementation

As our work was done on the basis of the TreadMarks' framework, we first describe briefly TreadMarks then outline the implementation details of our ARC model in sequence.

3.1 Platform

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) [6]. TreadMarks implements intermachine communication by using the Berkeley sockets interface and employs a **SIGIO** signal handler to deal with incoming requests, where message (whether a synchronization message or a data request message) arrival at any socket which is for receiving request messages generates a **SIGIO** signal. To implement the consistency protocol, TreadMarks uses the **mprotect** system call to control accesses to shared pages such that any process fetching a protected page generates a **SIGSEGV** signal, which triggers the segmentation fault handler to enforce data coherency. In order to overcome the coarse granularity and false sharing, TreadMarks propagates modifications (called *diffs*) instead of the whole updated pages, where a diff describes the modifications made to the page. In the remaining paper, unless explicitly specified, we use the diff request message and the diff message stand for the data request message and the data reply message, respectively.

Our experimental environment is based on the TreadMarks' framework, with most of its modules modified to accommodate the ARC model, where the **mprotect** system call and the **SIGSEGV** signal are well cooperated to enforce data coherence at the time when the processor fetches the first shared data after the lock acquire instruction, as can be seen from the detailed pseudo codes for implementing our ARC model given below.

3.2 Implementation Details

For any DSM, a layer of shared memory abstraction is constructed on top of the distributed memory system so that programmers can envision a globally shared address space. To do so, DSM normally provides two types of interface functions for communicating with the application program: (1) primitive functions such as Lock_acquire(), Lock_release(), and so on, to be called by the application program, and (2) interrupt handler routines to be invoked by such events as memory segmentation faults, the arrival of lock request or data request messages, and so forth. A type (1) function is engaged when the application program explicitly calls these functions, while a type (2) routine is triggered at the current processor when the application program fetches a protected memory area (leading to a segmentation fault) or at a remote processor when a lock (or data) request message reaches that remote processor.

ARC enforces data coherence at the time when a processor accesses the first shared data after a lock acquire instruction. To implement ARC, the lock request message is not sent at the lock acquire time, but instead, at the first shared memory access after the lock acquire, as described below. An overall diagram to illustrate the flow of ARC functions is provided at the end of this subsection.

Every processor under ARC does not actually send the lock request message out when executing Lock_acquire() (as illustrated in Fig. 2); instead, it executes the following two operations: (1) keeping the requested lock number in the variable *req_lockn* (see the 3rd last line in Fig. 2), and (2) setting all pages allocated by the application program to be not readable nor writable (by using **mprotect** system call). Note that the application program must allocate shared data before accessing it (as an example, through the Tmk_malloc() function in TreadMarks), and therefore, the underlined DSM system knows (at the shared memory allocation time) exactly how many pages the application program has allocated. As all those allocated pages are contiguous, the overhead of setting them to the same protection mode (i.e., the non-readable/writeable mode) involves only a single mprotect call.

```
Lock_acquire(I) {      /* processor requests lock I */
   if (req_lockn != FALSE) { /* any pending lock request? */
      send req_lock(req_lockn) to lock holder; /* to acq. lock */
      wait until receiving lock;
   }
   req_lockn = I;  /* mark down request lock number */
   set all allocated pages to be not readable nor writable;
}
```

Fig. 2. Lock acquire.

As all allocated pages are disabled in operation (2) above, any subsequent shared memory access (by the application program) invokes an interrupt (i.e., via the **SIGSEGV** signal), which activates the segmentation fault handler, as depicted in Fig. 3. At this moment, the lock request message piggybacked with the accessed page number (i.e., req_lock(*req_lockn*, *Y*)) is sent to the lock holder (see the 3th line in Fig. 3) to acquire the lock and the desired diff(s), if any. After the requested lock is received, variable req_lockn is reset to "FALSE" and the protection mode of each allocated page is restored to its previous state. As most contiguous pages tend to have the same protection mode due to the property of *locality of reference*, the overhead of restoring those pages is not significant, using a single mprotect call to reset a large chunk of pages. When compared with LRC, our ARC may in fact save one mprotect call per lock acquire, as stated in the following. The lock requestor in LRC, after receiving the lock reply message, invalidates those pages that are updated by the other processors, according to the write notices piggybacked in the lock reply message, with the invalidation of a page done by a mprotect call. A subsequent access toward any invalidated page activates the segmentation fault handler, Segv_handler(), to acquire the desired diff(s). In contrast, our ARC postpones the enforcement of data coherence at the time of the *first shared memory access* after the acquire, piggybacking the write notice and diff(s) in the diff message, and consequently, requiring no invalidation toward the first fetched page after the acquire. This saves one mprotect call for that fetched page.

The segmentation fault handler could be invoked without any pending lock request in existence, and if so, no lock request message is issued.

```
Segv_handler(Y)) {  /* Segmentation fault at page Y */
   if (req_lockn != FALSE) {      /* any pending lock request */
      send req_lock(req_lockn, Y) to lock holder;
      wait until receiving the lock;
      req_lockn = FALSE          /* reset variable */
      reset each allocated page to its previous state;
      incorporate write notices (and diff(s), if any);
   }
   handle regular segmentation faults;
}
```

Fig. 3. Segmentation fault handler.

The above scenario considers only the general case. In fact, a Lock_acquire(*I*) function (requesting for lock *I*) in an application program is possibly followed immediately by another Lock_acquire(*J*) function (requesting for lock *J*) or by a Barrier(*I*) function. Under such a situation, the second Lock_acquire() function (and the Barrier() function) must send out the previous pending lock request message immediately without piggybacking any page number, as shown in Fig. 2 (and Fig. 4). A future ARC implementation will exploit the potential described in Section 2.3 to save more messages.

```
Barrier(I) {   /* processor X requests barrier I */
   if (req_lockn != FALSE) { /* any pending lock req.? */
      send req_lock(req_lockn) to lock holder; /* to acq. lock */
      wait until receiving lock;
      req_lockn = FALSE      /* reset variable */
   }
   do barrier synchronization stuff;
}
```

Fig. 4. Barrier.

To get a lock, the processor issues a lock request message to the lock holder (i.e., a remote processor), and the arrival of a lock request message (at the remote processor) invokes the lock request handler, Lock_req_handler(), at that processor, as illustrated in Fig. 5. The received lock request message is forwarded to the lock holder (see the 2nd last line in Fig. 5) if the processor which receives the request does not hold the lock; otherwise, two possible situations occur, as follows. (1) If the processor is not in use of the lock, the lock (and possibly piggybacked with the requested diffs) is consigned to the lock requestor immediately (lines 4–7); otherwise, (2) the requested lock number (and the requested page number, if any) is recorded (the 5th last line) for later use at the lock release time. That is, the processor will send the lock release message to the requestor later on by executing Lock_release(*I*), as illustrated in Fig. 6.

```
Lock_req_handler(I) {      /* a request for lock I */
   if (processor holds lock I) {
      if (not in use of lock I) {
         if (request for a page and diff(s) of that page exists)
            send lock(I) with write notice(s) and diff(s);
         else
            send lock(I) with write notice(s);
      }
      else            /* processor is in use of lock I */
         record requested lock number (and page #, if any);
   }
   else            /* lock I is not in processor */
      forward request to the holder of lock I;
}
```

Fig. 5. Lock request handler.

```
Lock_release(I) {      /* processor releases lock I */
   if (req_lockn != FALSE) { /* any pending lock request? */
      send req_lock(req_lockn) to lock holder; /* to acq. lock */
      wait until receiving lock;
      req_lockn = FALSE      /* reset variable */
   }
   if (there has been a lock request) {
      if (request for a page and diff(s) of that page exists)
         send lock(I) with write notice(s) and diff(s);
      else
         send lock(I) with write notice(s);
   }
}
```

Fig. 6. Lock release.

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 lock requestor, if any; otherwise, the lock privilege stays there until it is requested later on. The diff(s) of a requested page might be piggybacked in the lock release message, as long as (1) the diff(s) exists at the lock releaser's site, and (2) the received lock request message contains a page number. Since the lock releaser generates the write notice(s) for the requestor, it knows, based on the requestor's *vector timestamps* [13] in the received lock request message, which diff(s) under what time intervals to be piggybacked.

Under a rare situation where no shared memory access exists between a lock acquire and a lock release, Lock_release() function then must send out the pending lock request message immediately without piggybacking any page number (see lines 2–6 in Fig. 6).

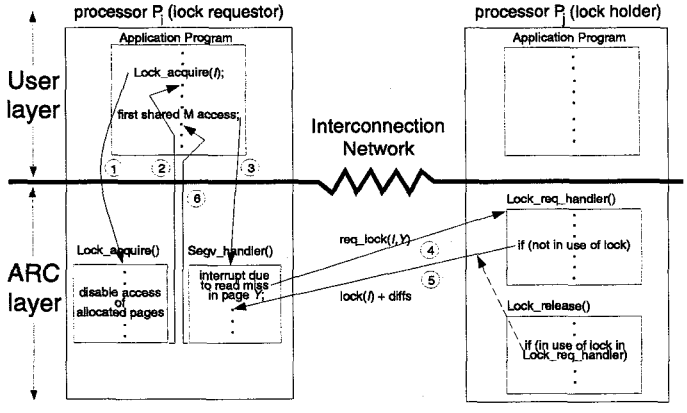

Fig. 7. Interactions between an application program and ARC functions.

A diagram to illustrate how these ARC interface functions interact with the application program is given in Fig. 7, where function calls, interrupt invocations, and the messages transmitted over the network are annotated with sequence numbers, and they occur in the order specified by the sequence number. For better explanation, a general case with only one lock followed by a read miss is shown here. The application program in processor P_i, on executing the Lock_acquire(*I*) function (to get lock *I*), causes all the allocated shared pages to be disabled and the requested lock number (i.e., *I*) to be recorded. After returning from the Lock_acquire() function, the application program continues executing the subsequent instructions until the first shared memory access (i.e., a read miss in page *Y* in this case) is encountered. As all allocated pages have been disabled, this shared memory access invokes an interrupt (i.e., a **SIGSEGV** signal) which activates the segmentation fault handler, Segv_handler(), and consequently a combined message (of the lock request message and the diff request message), req_lock(*I*, *Y*), is sent to the lock holder, P_j, through the interconnection network. The arrival of this combined message at P_j triggers a **SIGIO** signal which launches the lock request handler, Lock_req_handler(), with the following two possible cases in consigning the lock privilege through a release message: (1) a lock release message, lock(*I*), piggybacked with the diffs, is sent by the Lock_req_handler() to the lock requestor, P_i, if P_j is not in use of the lock; otherwise, (2) it is consigned by the lock release function, Lock_release(), using a lock release message piggybacked with the diff(s) to P_i. Eventually, the Segv_handler() routine in P_j returns control to the application program after receiving the lock release message and incorporating the write notices and diffs.

4 Benchmark Programs

Three application programs are employed to evaluate

the behavior of our proposed ARC model: QuickSort (QS), Integer Sort (IS), and Traveling Salesman Problem (TSP). As the ARC model enforces data coherence with respect to only the lock primitive, no application without locks is selected. In fact, we ran FFT (a benchmark program without any lock involved, from [9]) and found that the same performance was exhibited under both LRC and ARC, as expected. It is possible to incorporate ARC in the barrier primitive as well, in addition to the lock primitive as presented here. TSP uses lock primitives only, while the other two involve both lock and barrier primitives for synchronization. IS comes from the NAS benchmark suite [10], while QS and TSP are developed at Rice University [8].

4.1 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 dequeued unsorted elements. The entire array is inserted in a task queue initially. According to the quicksort, a dequeued unsorted array 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 any partition size reaches a threshold of 1K integers, at which time the partition is sorted locally by a bubblesort algorithm.

An array of 256K integers is considered in this study.

4.2 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 applied 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, 20 rankings were performed in each run with the values of N and B_{max} set to 2^{20} and 2^9, respectively.

4.3 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*, 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 [8] 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 employed in this study.

5 Performance Evaluation

Our 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. This is a typical network of workstations (NOW) platform. The page size is set to 4096 bytes and the invalidate protocol is employed to enforce data coherence. In an invalidate protocol, the acquiring processor invalidates those pages in its cache according to the received write notices.

Performance results for the three benchmark programs under LRC and ARC are compared below.

5.1 Results

Performance results of LRC and our proposed ARC are provided in Table 1 and Figs. 8–10, where the results are averaged over 30 independent runs for each application program. Five parameters reflecting the speciality of ARC are listed in Table 1: the number of diff messages and the amount of diff data which are piggybacked in lock release messages (in our ARC model), the number of diff request messages and the total data amount of all diff request messages, and the averaged size of a message.

Table 1. Performance results of LRC and ARC

Prog	Alg	Piggy-backed diff msg	Piggy-backed diff data amount (Kbytes)	Diff request msgs	Diff request data amount (Kbytes)	bytes per msg
QS	LRC	0	0	3256	80.6	899
	ARC	1148	271	2122	40.7	1065
IS	LRC	0	0	299	7.4	1179
	ARC	139	1116	164	3.6	1223
TSP	LRC	0	0	3146	66.7	254
	ARC	430	482	2649	50.8	284

It is seen from the table that ARC indeed experiences ample chances of combining lock release messages and diff messages, as a result of deferring the issue of lock request messages until the first access to a shared location after the acquire instruction, for all the three application programs. As an example, 1148 diff messages are piggybacked in the lock release messages for QS. Similarly, ARC also effects the mergence of many lock request messages and diff request messages, leading to reduction in the total number of diff request messages and the data amount of diff request messages. A message size increase is expected for ARC as it merges a lock request (or release) message and a diff request (or a diff) message into one single message. This table does not list explicitly figures related to the lock request messages, as both LRC and ARC have similar results in terms of the message

number and the data amount. Note that the lock request message has a fixed size, with a message header plus the respective requested lock number, while the diff request message has a varying size, with a message header and the requested page number plus information about which diff(s) to be retrieved. The combination of a lock request message and a diff request message in ARC saves both one message header and above mentioned information about diff(s). The number of diff messages (which equals the number of diff request messages) and the amount of total diff data are shown in Fig. 8.

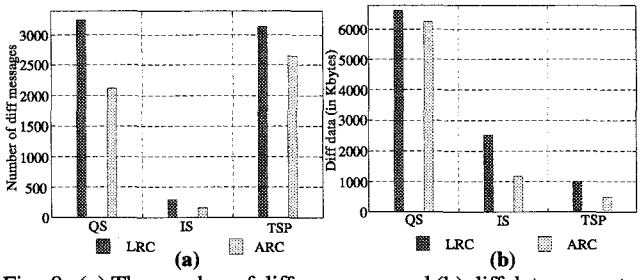

Fig. 8. (a) The number of diff messages and (b) diff data amount.

ARC attempts to lower the total number of messages transmitted over the network by piggybacking the diff data in a lock release message whenever possible, leading to significant reduction in the number of diff messages and the amount of diff data, as depicted in Fig. 8. It can be observed from the figure that ARC reduces the respective number of diff messages by 35%, 45%, and 16%. Correspondingly, ARC saves the amount of diff data by 6%, 54%, and 50% for QS, IS, and TSP, respectively, as shown in Fig. 8(b). ARC always leads to significant reduction in the number of diff messages over LRC for all the three programs examined, revealing its effectiveness. QS is found to have less improvement in diff data than IS (and TSP), because it has a finer granularity with respect to diff data than IS. Specifically, QS accesses up to 1026 shared pages on the network of 8 workstations and only a few bytes are updated in most pages at the time of data coherence enforcement. In contrast, IS accesses only one page for the given data size throughout its life time, with all updates done in that page.

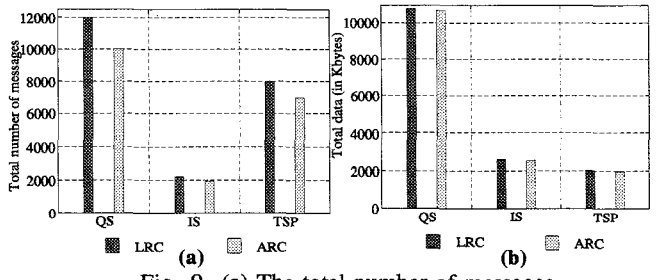

Fig. 9. (a) The total number of messages and (b) total data amount.

The total number of messages for each program is depicted in Fig. 9(a), where ARC consistently gives rise to fewer total messages. This reduction in the total number of messages is due mainly to a reduced number of diff messages and diff request messages, and it is up to 16.5%, 12.3%, and 12.7% for QS, IS, and TSP, respectively. On the other hand, ARC gains limited reduction in the total data amount, only by 1.1%, 1.2%, and 2.2%, respectively, as illustrated in Fig. 9(b). ARC indeed generates lighter coherence traffic over the network (due chiefly to reduction in the total number of messages), as might be expected.

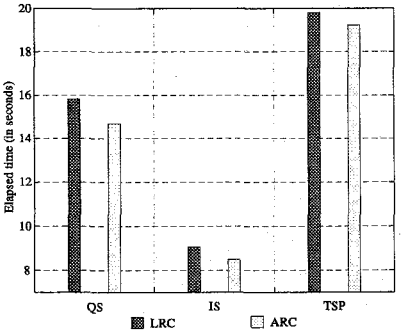

Fig. 10. Total elapsed time.

For a software DSM system, such as our NOW platform, lower traffic over the network is particularly profitable. In general, communication latency consists of three components: start-up latency, network latency, and blocking time. Start-up latency is the time required to handle the message at both the source and destination nodes. Network latency equals the elapsed time after the head of a message has entered the network at the source until the tail of the message emerges from the network at the destination. The blocking time is due to network contention. With a fixed amount of data, it is advantageous to deliver the data through fewer messages, which save network start-up latency and decrease the degree of network contention. As a result, the communication latency for delivering a message is not proportional to the message length. For example, sending a message of 8 bytes over ATM AAL3/4 sockets takes 80 μsec, while it is only 1367 μsec to deliver 9188 bytes, as stated in [8]. Reduction in the total number of messages due to ARC therefore has a positive impact on the execution time, as can be seen in Fig. 10, where ARC promises speedier execution than LRC for every benchmark examined. The elapsed time savings are 1.2 seconds, 0.5 second, and 0.6 second, respectively, for QS, IS, and TSP.

6 Pertinent Work

Two related memory consistency models are stated and contrasted with our ARC in sequence.

6.1 Lazy Hybrid Protocol

A variant protocol of LRC, called the *lazy hybrid protocol* (LH for short) was considered earlier by Keleher [8]. LH speculatively moves data before it is requested by the next processor to reduce the number of diff messages. The basic philosophy behind LH is that the flow of data is likely to mirror the flow of synchronization, as processes must synchronize in order to fetch shared data. As a result, LH uses a heuristic method on *approximate copyset* to track accesses to shared pages (by other processes) in

order to decide which pages are most likely to be used, and those diffs associated with the predicted pages are then appended to the lock release message, or are flushed to all other processors prior to barrier arrivals.

Compared with our ARC model, LH has the following main drawbacks: (1) the heuristic method employed in LH cannot guarantee a high hit rate, and consequently (2) any misprediction by the heuristic method will piggyback extra diffs (associated with the mispredicted pages) which are useless, thus increasing the total amount of data transferred over the network and resulting in a longer program execution time. In fact, the experimental results for all application programs examined in [8] indicate that LH consistently transfers more total data than LRC. In some cases, LH even leads to more total messages and/or longer program execution times than LRC. In contrast, as our ARC model enforces data consistency at the time of the first shared memory access after the acquire, (1) it always predicts the required page correctly and thus the piggybacked diffs in the lock message are surely to be used by the lock requestor, and (2) no processor under ARC delivers unnecessary diffs, yielding a smaller data amount, fewer messages, and a shorter program execution time.

6.2 Entry Consistency

Unlike LRC, which enforces data coherency at the page level, Entry Consistency (EC) [14], used in Midway [15], treats shared data in terms of objects. EC requires all shared data objects to be explicitly associated with synchronization variables (locks or barriers) such that it can identify which updates must be propagated. In other words, when a processor acquires a synchronization variable, the shared data associated with that synchronization variable is also propagated to the requestor. Since compilers cannot do this automatically today, however, the association of data objects with synchronization variables tends to impose a substantial programming burden on programmers. In contrast, our ARC model can identify correctly which updates to be propagated without any help from compilers or programmers, keeping the programming interface as simple as that of LRC.

7 Concluding Remarks

In this paper, we have proposed a new software-based DSM model, called *aggressive release consistency* (ARC), and have implemented ARC on a network of workstations for experimental performance evaluation. Our experimental environment is based on the TreadMarks' framework, with most modules in TreadMarks modified to accommodate our model, where TreadMarks is a software-based DSM implementation for lazy release consistency (LRC). It is observed from our experimental study using three benchmark programs that a lock request (or release) message and a data request (or reply) message due to memory coherence enforcement in our ARC model are likely to be merged into one message, consequently reducing the number of messages transferred over the network and therefore lowering the execution times of application programs. The experimental results demonstrate that the proposed ARC model gives rise to better performance than LRC.

Release consistency enforces data coherence at synchronization points in connection with locks or barriers. In this paper, we apply our ARC model only to the lock access, attaining noticeable performance improvement over LRC. In the future, we shall apply our ARC model to the barriers as well. Also, a larger system is expected to enjoy a higher performance gain resulting from ARC, because the number of messages transmitted over the network then tends to increase. It is interesting to carry out an experimental study of our ARC model on a medium-sized machine, like the IBM SP2, and evaluate its performance as the system size grows.

Finally, the number of instructions between a lock acquire (or barrier) instruction and its subsequent, first shared memory access instruction dictates how late the lock request message will be issued under ARC. Earlier issuance of a lock request leads to the lock to arrive sooner, but has the possible disadvantage of generating more diff messages and more transferred data. It is interesting to investigate how many instructions thereinto may result in the best benefit, and this information could be useful for instruction scheduling to optimize system performance.

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, vol. 15, pp. 54-64, Feb. 1995.

[2] N.-F. Tzeng and P.-C. Yew, "Special Issue on Distributed Shared Memory Systems," *J. Parallel and Distributed Computing*, vol. 29, No. 2, Sept. 1995.

[3] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel & Distributed Technology*, vol. 4, pp. 63-79, Summer 1996.

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

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

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

[7] C. Amza *et al.*, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computers,* vol. 29, pp. 18–28, Feb. 1996.

[8] P. Keleher, *Lazy Release Consistency for Distributed Shared Memory,* Ph.D thesis, Rice University, Jan. 1995.

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

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

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," *Proc. 13th ACM Symp. Operating Systems Principles*, pp. 152-164, Oct. 1991.

[12] K. L. Johnson *et al.*, "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 213–228, Dec. 1995.

[13] P. Keleher *et al.*, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. the Winter USENIX Conference,* pp. 115–132, Jan. 1994.

[14] B. N. Bershad *et al.*, "The Midway Distributed Shared Memory System," *Proc. IEEE CompCon Conference,* pp. 528–537, Feb. 1993.

[15] M. J. Zekauskas *et al.*, "Software Write Detection for Distributed Shared Memory," *Proc. 1st Symp. on Operating Systems Design and Implementation (OSDI),* pp. 87–100, Nov. 1994.