Logging and Recovery in Adaptive Software Distributed Shared Memory Systems

Angkul Kongmunvattana and Nian-Feng Tzeng

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

Abstract

Software distributed shared memory (DSM) improves the programmability of message-passing machines and workstation clusters by providing a shared memory abstract (i.e., a coherent global address space) to programmers. As in any distributed system, however, the probability of software DSM failures increases as the system size grows. This paper presents a new, efficient logging protocol for adaptive software DSM (ADSM), called adaptive logging (AL). It is suitable for both coordinated and independent checkpointing since it speeds up the recovery process and eliminates the unbounded rollback problem associated with independent checkpointing. By leveraging the existing coherence data maintained by ADSM, our AL protocol adapts to log only unrecoverable data (which cannot be recreated or retrieved after a failure) necessary for correct recovery, reducing both the number of messages logged and the amount of logged data.

We have performed experiments on a cluster of eight Sun Ultra-5 workstations, comparing our AL protocol against the previous message logging (ML) protocol by implementing both protocols in TreadMarks-based ADSM. The experimental results show that our AL protocol consistently outperforms the ML protocol: Our protocol increases the execution time slightly by 2% to 10% during failure-free execution, while the ML protocol lengthens the execution time by many folds due to its larger log size and higher number of messages logged. Our AL-based recovery also outperforms ML-based recovery by 9% to 17% under parallel application examined.

1. Introduction

Parallel programming under the message-passing paradigm on distributed memory systems, like workstation clusters, has been known to be difficult because it involves interprocess communication operations and requires explicit data partitioning. Software distributed shared memory (DSM) simplifies parallel programming tasks by providing a shared memory abstract (i.e., a coherent global address space) to programmers, making cluster computing attractive. This shared memory abstraction spans across memory modules at interconnected nodes. Memory coherence is maintained through manipulating a virtual memory protection mechanism, which is readily available in most, if not all, commodity microprocessors, without any extra hardware. Software DSM performance is dictated by the number of messages exchanged and the amount of data transfer over the networks. In most cases, memory consistency models and coherence enforcement protocols play a significant role on both the communication frequency and the load of communication traffic. Software DSM often adopts a relaxed memory consistency model [1], which makes several optimizations possible. The basic version of software DSM, dubbed *Basic DSM* (BDSM), uses write-invalidate (WI) and multiple-writer (MW) protocols for enforcing its memory coherence [3]. To improve performance further, state-of-the-art software DSM also employs write-update (WU) and single-writer (SW) protocols as alternatives. This advanced version of software DSM, referred to as *Adaptive DSM* (ADSM), adapts its coherence enforcement protocols according to shared memory access patterns of the application programs [2, 15]. While faster processors, higher network bandwidth, and more sophisticated protocols continue to improve software DSM performance and scalability, the probability of system failures also increases as the system size grows. Hence, a mechanism for supporting fast crash recovery in software DSM is required [20], giving rise to *recoverable software DSM systems*.

Message logging is a popular technique for providing BDSM with fault-tolerant capability [7, 14, 16, 17, 19]. This technique is attractive because it allows independent checkpointing without any domino effect and improves the crash recovery speed when coordinated checkpointing is employed [9]. While those earlier logging protocols work well under BDSM, they cannot be directly applied to ADSM. This is because ADSM maintains its coherence enforcement data differently from BDSM. Specif-

ically, BDSM always keeps a summary of modifications made to each shared memory page (known as *diff*), while ADSM creates diff only when the MW protocol is used for enforcing the coherence of shared memory pages. Although a message logging protocol without leveraging any coherence enforcement data maintained by software DSM was considered [19], several later studies have shown that such a naive implementation is undesirable due to its high overhead during the failure-free execution [7, 16].

In this paper, we propose a new, efficient logging protocol, called *adaptive logging* (AL), for ADSM. Our AL protocol closely tracks the adaptation of coherence enforcement for each shared memory page and only records information indispensable for recovery, insuring correct recovery with minimum overhead. To assess the impacts of our AL protocol on ADSM performance, we have implemented our AL protocol in TreadMarks-based ADSM [15] and conducted an experimental study. We also implemented an earlier message logging (ML) protocol [19] separately in TreadMarks-based ADSM for comparison. Our experiment has been conducted on a cluster of eight Sun Ultra-5 workstations. Results demonstrate that our AL protocol consistently outperforms the ML protocol: Our protocol increases the execution time merely by 2% to 10% during failure-free execution, while the ML protocol lengthens the execution time from threefold to more than eightfold (i.e., 280% to 840%) due to its large log sizes. Our recovery also performs better than ML-based recovery by 9% to 17% under parallel applications examined.

The outline of this paper is as follows. Section 2 provides basic background pertinent to this work and details on some related work. Our adaptive logging protocol along with the checkpointing and recovery techniques is described in Section 3. Section 4 presents our experimental setup and parallel applications used in this study. Performance results of our proposed protocol are demonstrated in Section 5. We conclude this paper in Section 6.

2. Pertinent Background

2.1 Software DSM

Software DSM creates a shared memory image on top of parallel systems with physically distributed processing nodes, such as distributed memory multiprocessors and workstation clusters. These processing nodes (i.e., processors) are connected via high-speed networks, such as the fast Ethernet, ATM, or VIA. A group of processes run on such nodes to execute a parallel application program, and they enforce memory coherence through explicit message exchange. The memory coherence enforcement protocols are often devised under the notion of relaxed memory consistency models [1], aimed at low coherence traffic and good performance. To this end, release consistency [10] (i.e., one of the least restrictive relaxed memory consistency models) is a favorite choice because it does not guarantee that shared memory is consistent all of the time, but rather making sure consistence only after synchronization operations (i.e., locks and barriers). In essence, release consistency ensures a synchronized program to see a sequentially consistent execution through the use of two synchronization primitives: *acquire* for a process to get access to the shared variable, and *release* for a process to relinquish an acquired variable, permitting another process to acquire the variable. In a synchronized program, a process is allowed to use a shared data only after acquiring it, and the acquired data may then be accessed and modified before being released (and subsequently acquired by another process). Each process is also permitted to update the shared data multiple times locally, but all the updates have to be completed before the release is performed.

Lazy release consistency (LRC) [13] is an efficient software implementation of the release consistency model. It postpones coherence enforcement until the acquire is performed (by another process). Instead of sending coherence information to all other processes at a release, LRC allows coherence information to be piggybacked on the lock grant message sent to the process at which acquire is performed, reducing the number of messages needed for enforcing memory coherence. LRC also avoids sending messages unnecessarily to those processes which do not require coherence information. As a result, LRC-based software DSM generally involves fewer messages and less data exchanged than other software DSM implementations. There are several basic memory coherence enforcement protocols available under LRC implementation. Each protocol demands a different amount of resources (i.e., memory) and generates a different traffic amount (i.e., messages) over the networks, yielding different performance gains for different shared memory access patterns. We explain each protocol under the context of LRC as follows:

- **Multiple-Writer Protocol (MW)** alleviates the thrashing problem due to falsely shared data (i.e., reduces traffic over the network) by allowing each process to concurrently modify non-overlapping parts of its local copy of the same shared memory page. Since these modifications have to be accumulated and merged at the next synchronization point, it consumes a significant amount of memory and induces extra computation.

- **Single-Writer Protocol (SW)** has very low protocol overhead but permits only one writable copy of any shared memory page at any given time. Although LRC-based SW allows multiple read-only copies to co-exist with one writable copy, it often creates an excessive amount of data transfer on write-write false sharing data.

- **Write-Update Protocol (WU)** enforces memory coherence by updating all copies of modified shared memory pages at a synchronization point. Modification information is piggybacked with synchronization messages. With the MW protocol, only the summary of modifications is sent, instead of a whole memory page (as in the SW protocol). In both cases, the WU protocol often sends a substantially large amount of data to processes that do not need them.

- **Write-Invalidate Protocol (WI)** invalidates all other copies of shared memory pages that have been modified. Invalidation is performed at a synchronization point. Only the processes that actually need the up-to-date copy of these pages will request for the updates, optimizing the data transfer amount at the expense of memory access miss latency.

In the basic implementation of software DSM, called BDSM, the consistency of shared memory pages is enforced by MW and WI protocols, which optimize both the data transfer amount and the number of message exchange [3]. Past research in message logging for software DSM is based on this BDSM implementation, where two static coherence enforcement protocols, MW and WI, are applied to all shared memory pages [7, 14, 16, 19]. While BDSM works well on several applications, recent studies have shown that flexibility on selecting a coherence enforcement protocol for each shared memory page can improve software DSM performance significantly [2, 15]. Next, we explain such an adaptive approach in detail.

2.2 Adaptive DSM (ADSM)

ADSM refers to an LRC-based software DSM system with multiple basic protocols employed [2, 15], and it dynamically determines the protocol operations for each shared memory page during run-time, according to both data transfer amounts and shared memory access patterns. Such adaptation in ADSM allows great flexibility and improves software DSM performance on a wide range of applications because it selects appropriate coherence enforcement protocols for each shared memory page, reducing coherence enforcement overhead [2, 15]. The sharing patterns can be classified as follows:

- **Migratory Data** refers to shared memory pages that are guarded by lock synchronization primitives, permitting only one processor to access the protected data at a time. The ownership of these shared pages is transferred from one processor to another at lock release and acquire, hence the name migratory. Consistency of shared memory pages in this class is enforced by the SW protocol because no more than one processor may modify these pages at a time. The WU protocol is applied only at lock acquire where access patterns are well-predicted; otherwise, the WI protocol is used.

- **One Producer/Multiple Consumers Data** refers to shared memory pages that are repeatedly modified by the same processor, and subsequently consumed by other processors. Shared memory pages in this category are guarded with barrier synchronization primitives and are managed by SW and WU protocols. This is appropriate since the same processor always holds the ownership of these pages. While the use of an update protocol at the barrier causes a substantial amount of data transfer, it reduces memory miss idle time for well-predicted access patterns, improving software DSM performance.

- **Falsely Shared Data** refers to shared memory pages that are modified by multiple processors at different parts of their contents at the same time. Shared memory pages in this case are enforced by the MW and WI protocols. Without the MW protocol, simultaneous access to the same shared memory page is not permitted and the page will bounce among a group of processors that wish to access it, leading to a thrashing problem. The invalidation protocol is applied both at the lock acquire and at the barrier, and therefore, each processor will receive data only when it is needed, minimizing the data transfer amount and coherence traffic.

Coherence protocol adaptation in LRC-based software DSM was considered first by Dwarkadas $et\ al.$ [8], who introduced a hybrid protocol based on the integration of WI and WU protocols. Their simulation results demonstrated that the benefits of WI and WU protocols can be combined, reducing the data transfer amount as well as the number of memory access misses and of messages exchanged. Later on, Amza $et\ al.$ [4] studied the benefits of software DSM protocols that adapt between SW and MW protocols. They used the write granularity and the false sharing effect to select the protocols. Specifically, the SW protocol was used with little or no write-write false sharing data, whereas the MW protocol was employed for falsely shared data. They concluded that the adaptive protocols consume less memory than the MW protocol and generate lower communication traffic than the SW protocol. Recently, Monnerat & Bianchini [15] and Amza $et\ al.$ [2] investigated the adaptation between SW and MW protocols, and between WI and WU protocols. They explored adapting these protocols in response to the changes of shared memory access patterns. Their results demonstrated the benefits of adaptive protocols in software DSM, leading to substantial performance improvement. These adaptive protocols serve as a basis of our study. Next, we summarize earlier work on message logging for software DSM.

2.3 Related Work

Message logging has its root in message-passing systems and has been studied extensively [6, 9, 11, 12, 18]. While it is possible to apply these protocols directly to LRC-based software DSM [19], a system resulting from this direct application is undesirable because of high overhead during failure-free execution [7, 16].

The study on message logging for software DSM has been attempted by researchers [17, 19]. In particular, Suri, Janssens, and Fuchs proposed a logging technique for LRC-based software DSM with MW and WI protocols employed (i.e., BDSM) [19]. Their technique logs every incoming message related to the state of shared data and does not leverage coherence data maintained by BDSM. Unfortunately, this technique has been proven to cause excessive overhead, hampering BDSM performance [7, 16]. Costa *et al.* considered a vector clock logging technique [7] in which the logged data are kept in volatile memory of the sender process, leveraging the coherence data present in BDSM. Such a logging technique consequently keeps much less data, but it cannot handle multiple failures. A logging technique obtained through refining the technique proposed in [19] by logging only the contents of lock grant messages (i.e., the list of dirty pages) is known as reduced-stable logging (RSL) [16]. Simulation results indicated that RSL utilized less disk space for the logged data than the technique given in [19]. Recently, lazy logging has been devised [14], and experimental results have shown that lazy logging causes lower execution overhead than RSL because of the significantly reduced log size and number of accesses to stable storage.

While all these earlier message logging protocols work well under BDSM, only those which do not rely on coherence data maintained by BDSM can be applied to ADSM, an adaptive multi-protocol software DSM, but those logging protocols always yield high overhead and are thus undesirable. We propose an efficient logging protocol, which tracks closely the adaptation of coherence enforcement, for ADSM in the next section, and compare the execution overhead of our adaptive logging protocol with that of a previous approach in Section 5.

3. Proposed Adaptive Logging

3.1 Motivation

Traditionally, logging protocols for software DSM are proposed for unbounded rollback recovery of independent checkpointing techniques [7, 14, 16, 17, 19]. Without modifications, these logging protocols can also speed up the recovery of coordinated checkpointing techniques [9].

An efficient logging protocol for software DSM leverages existing coherence data maintained by software DSM itself [7, 14, 16]. Such a logging protocol requires to understand coherence enforcement protocols adopted by each software DSM system. In particular, prior logging protocols are designed for software DSM that manages all of its shared memory pages using the MW and WI protocols (i.e., BDSM), and therefore, those logging protocols assume that the sender process always maintains a copy of diff (i.e., a summary of modifications) for each shared memory page in its memory. As software DSM continues to improve its performance by implementing an increasingly sophisticated memory coherence protocol, this assumption no longer holds true. ADSM, for example, employs a wide range of coherence enforcement protocols and manages each shared memory page according to its access patterns [2, 15]. This improvement renders the assumption made by earlier logging protocols obsolete since the sender process then no longer always maintains the coherence data in its memory. While a trivial approach is to resort to a logging protocol that makes no use of coherence data maintained by software DSM [19], we demonstrate that such an approach is undesirable due to its excessive overhead during failure-free execution (see Section 5 for experimental results), and thus, there is a need to devise a suitable logging protocol for ADSM.

3.2 Protocol Description

Our adaptive logging (AL) protocol aims at providing low-overhead crash recovery capability to ADSM. It logs only the coherence data that is truly necessary for correct recovery (i.e., cannot be retrieved or reconstructed during recovery) in ADSM. In addition, our AL protocol also creates a volatile log of such coherence data at the sender process that is not maintained by ADSM. Coherence-related messages in ADSM can be classified as follows:

- **Write-Invalidation Message (WIM)** refers to a message that contains a list of dirty pages (i.e., write-invalidation notice), and that exclusively utilizes by the WI protocol. This type of messages is sent only at a synchronization point and is typically small (less than 2KB). At a lock acquire, WIM is referred to as a lock grant message since it not only contains a write-invalidation notice, but also allows a receiving process (i.e., an acquirer) to enter the critical section. At a barrier, the barrier manager receives WIM from all participating processes, as check-in messages, and sends out WIM to all processes as check-out messages.

- **Write-Update Message (WUM)** refers to a message that carries either a summary of modifications (i.e., a diff) or an up-to-date copy of shared memory pages, and that is solely used by the WU protocol. This type of messages is also sent only at a synchronization point. Its size is usually larger than that of WIM since it contains either diffs or copies of whole memory pages. At

a lock acquire, a lock grant message is piggybacked with diffs when the MW protocol is selected; otherwise, it is piggybacked with up-to-date copies of shared memory pages when the SW protocol is used. At a barrier, the barrier manager also receives WUM from all other processes as check-in messages, and sends out WUM to every process as check-out messages.

- **Memory-Update Message (MUM)** refers to a message that delivers either a summary of modifications (i.e., a diff) or an up-to-date copy of a shared memory page. This type of messages is used for updating a local copy of an invalid shared memory page, and therefore, only pages managed by the WI protocol incur MUM. The size of MUM ranges from a few bytes to 8KB (equal to the size of a shared memory page), depending on the coherence protocol selected (i.e., a diff for MW and a whole page for SW).

For a logging protocol that does not take advantage of coherence data maintained by software DSM [19], all these coherence-related messages must be logged by the receiver, referred to as the message logging (ML) protocol. Such a naive implementation tends to induce an excessive log size, hampering ADSM performance (as will be demonstrated in Section 5). On the contrary, our AL protocol leverages existing coherence data maintained by ADSM and logs only WIM, keeping logging overhead low during failure-free execution. Since only WIM is logged and WIM is typically smaller than WUM and MUM, our AL protocol significantly reduces both the frequency of disk accesses and the amount of logged data. Logging WIM alone, however, cannot guarantee correct crash recovery because WUM and MUM for a shared memory page under the SW protocol do not create a copy of coherence data, and therefore, it is necessary for our AL protocol to produce a volatile log for such data at the sender process. Although the total size of our volatile log may be large, it has little impact on ADSM performance because such a volatile log is created and swapped out later whenever memory space runs short (and such a volatile log is never paged in again till the time of recovery). By contrast, coherence data created and maintained in memory by ADSM tend to be accessed and modified frequently, critical to overall performance. Whenever memory consumption of ADSM reaches a pre-set threshold, its storage reclamation routine will clean up the coherence data. Such a process involves a large amount of data transfer and many communication messages. We therefore overlap our garbage collection of the volatile log, which requires no interprocess communication, with the ADSM storage reclamation process, hiding extra overhead altogether.

In summary, our AL protocol leverages coherence data maintained by ADSM to reduce the number of message and

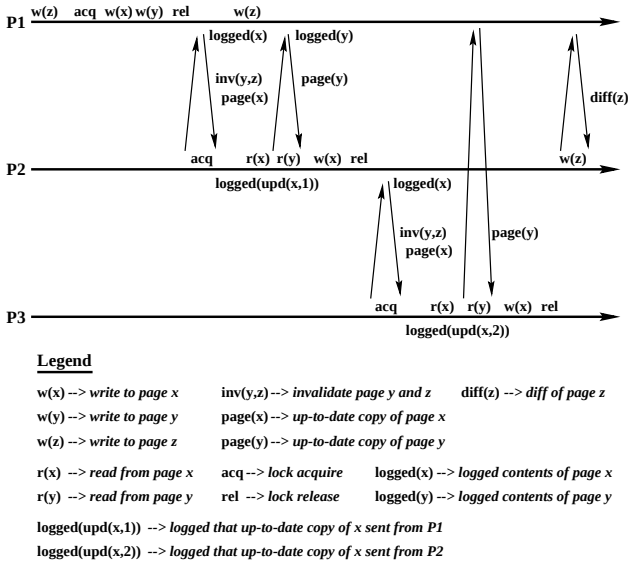

Figure 1. A Snapshot of Adaptive Logging and ADSM Protocols in Action.

the amount of coherence data to be logged. While it creates a volatile log whenever necessary, the volatile log so created has little impact on memory resource and ADSM performance because it is not accessed till the recovery time. Experimental results show that our AL protocol increases the execution time slightly, by 2% to 10% for parallel applications examined, as will be shown in Section 5. Next, we exemplify a snapshot of ADSM and our AL protocol.

Example

Figure 1 depicts a snapshot of adaptive logging in ADSM. In this example, shared memory pages x, y, and z are classified to be migratory, producer/consumer(s), and falsely shared, respectively. We assume that page x is migrated from process p_1 to process p_2, and then to process p_3. Page y is produced by process p_1 and subsequently consumed by both p_2 and p_3. Finally, page z is falsely shared by processes p_1 and p_2. Under this assumption, page x is managed by SW and WU protocols, page y is by SW and WI protocols, and page z is by MW and WI protocols. Apparently, a lock grant message from process p_1 to process p_2 (of an acquire) is piggybacked with the up-to-date copy of page x plus a list of dirty pages to be invalidated (i.e., pages y and z, in this example). Process p_1 creates a volatile log of page x before it makes any further modification to the page. Upon receiving a lock grant message from p_1, process p_2 not only updates its local copy of page x and invalidates pages y and z, but also records that it has received an up-to-date copy of page x from process p_1. Note that the WI protocol keeps a list of dirty pages along with a list of processes that modified those pages, so no extra record has to be kept for our recovery protocol (which utilizes the

same dirty page list). When process p_1 receives a request for page y from process p_2, it creates a volatile log of page y before sending out an up-to-date copy of page y to process p_2. On the contrary, upon receiving a memory update request of page z from process p_2, process p_1 simply creates a diff (i.e., a summary of modifications) and sends it to process p_2. No volatile log is created in this case because the MW protocol (while manages page z) keeps all diffs until the garbage collection routine is activated. For process p_3, on receiving a lock grant message from p_2, it creates a record which specifies its receipt of an up-to-date copy of page x from process p_2. Process p_2 has to create a volatile log of page x before making any further change to the page. When process p_3 requests for an up-to-date copy of page y from process p_1, p_1 need not to create another volatile log for page y since the previously created copy (i.e., the one generated when it sent page y to process p_2) is adequate for correct recovery under LRC.

3.2.1 Checkpointing

As we have mentioned earlier, our AL protocol is suitable for both coordinated and independent checkpointing since it speeds up the recovery process and avoids the unbounded rollback problem associated with independent checkpointing. Periodically, a checkpoint is created to limit the amount of work that has to be repeated after a failure. Under a coordinated checkpointing scheme, a checkpoint is taken at the selected synchronization points, where all processes exchange messages (i.e., barrier). For independent checkpointing, each process independently creates a checkpoint on stable storage. The difference between coordinated and independent checkpointing lies in how to reconstruct a globally consistent state of execution. Specifically, a globally consistent state can be rebuilt by combining the checkpoints of every process under a coordinated checkpointing technique, whereas the logged data is needed for a failed process in reconstructing a globally consistent state of execution under an independent checkpointing technique. While coordinated checkpointing may do without logging, the logged data speed up recovery by eliminating synchronization messages and reducing memory miss idle time during the crash recovery process. The selection of checkpointing techniques for ADSM, however, is beyond the scope of this paper.

In both cases, a checkpoint consists of all local variables and shared memory contents, the state of execution, and all local internal data structures used by ADSM. All such information is needed at the beginning of a crash recovery process to avoid restarting from the (global) initial state. While the first checkpoint flushes all shared memory pages to stable storage, our incremental checkpointing incorporates into the subsequent checkpoints, only those pages that have been modified since the last checkpoint. Specifically, our checkpointing protocol relies on coherence information and memory update logs to identify and selectively flush the changes to the shared memory pages into stable storage, reducing failure-free overhead.

3.2.2 Recovery

Our AL-based recovery is activated after a failure occurs and is detected. It reconstructs the execution prior to the failure by restarting from the most recent checkpoint and replaying the log of events. The log of events for our AL-based recovery contains lists of dirty pages to be invalidated or updated at each synchronization point. For ML-based recovery, the log of events includes all messages received at synchronization points and at memory misses. There is no orphan process created under either AL or ML protocols because all three message types (i.e., WIM, WUM, MUM) in ADSM are sent out only in response to request messages from other processes. During recovery, there will be no request for WIM since the recovery process can read a write-invalidation notice from its local logged data. The messages requesting for WUM and MUM do not create any orphan process because such requests do not change the status of shared memory pages at the receiving process, and therefore, it is unnecessary for surviving processes to rollback and "unreceive" these requests. As a result, no orphan process is created.

Since our AL protocol guarantees that all shared memory pages needed for memory update operations during recovery are readily available from the sender process(es), the recovery process can overlap such update requests with disk access operations and also benefits from light traffic over the network during the time of recovery, tolerating disk access latency. This is not possible in ML-based recovery because all of its logged data are then stored on the local disk. Both recovery techniques, however, benefit from the elimination of synchronization messages and the reduction of memory miss idle time. Experimental results demonstrate that AL-based recovery consistently outperforms ML-based recovery by 9% to 17% under parallel applications examined. Next, we present the implementation of our AL protocol and AL-based recovery in ADSM.

3.3 Implementation

Figure 2 demonstrates a pseudo code of the ADSM routines involved in our logging and crash recovery processes.

3.3.1 Failure-Free Execution

During failure-free execution, at a lock acquire, the sender process creates a volatile log of each shared memory page that is piggybacked with the lock grant message under the WU protocol. The receiver process records incoming coherence information as a list of up-to-date and dirty

Barrier

```
if (in_recovery)
    read list of dirty pages from log;
    invalidate the pages and/or send
        request for the up-to-date copy;
else
    if (barrier_manager)
        flush log to stable storage;
        wait for all other processes
            to check-in;
        invalidate or update local copy
            of page(s) correspondingly;
        send barrier release messages
            piggybacked with list of
            dirty pages and/or
            up-to-date copy of pages;
        record the pages that have received
            an up-to-date copy with an id
            of sender process;
    else
        send check-in message to
            barrier manager;
        flush log to stable storage;
        wait for barrier release message;
        invalidate or update local copy
            of page(s) correspondingly;
        record the pages that have received
            an up-to-date copy with an id
            of sender process;
    endif
endif
```

Lock Acquire

```
if (in_recovery)
    read list of dirty pages from log;
    invalidate the pages and/or send
        request for the up-to-date copy;
else
    send lock request to lock manager;
    flush log to stable storage;
    wait for lock grant message;
    invalidate or update local copy
        of page(s) correspondingly;
    record the pages that have received
        an up-to-date copy with an id
        of sender process;
endif
```

Lock Manager

```
if (receive_lock_request)
    send lock grant message
        piggybacked with list of dirty
        pages for pages under WI
        protocol and an up-to-date
        copy of pages for pages under
        WU protocol;
    log contents of those up-to-date
        copies of pages;
endif
```

Manager of Memory Page(s)

```
if (receive_page_request)
    send an up-to-date copy of that page to requesting process;
    log contents of an up-to-date copy of that page;
endif
```

Note: Bold fonts refer to ADSM protocol operations, while italic ones refer to operations of our Adaptive Logging and AL-based Recovery Protocols.

Figure 2. Pseudo Code for Adaptive Logging and AL-based Recovery Protocols.

pages, and then proceeds to lock acquire and coherence enforcement routines. Each process also produces a volatile log of shared memory pages it sends out upon receiving a page request message. While a list of shared memory page status has to be flushed to stable storage whenever a process sends a lock grant message to another process, the volatile logs need not to be flushed since the sender process can recreate them even after a process failure occurs.

At a barrier, the producer process of the producer/consumer pages creates a volatile log of shared memory pages it sends to the consumer process(es). This volatile log is created right after the barrier check-in messages have arrived and actual shared memory pages have been sent off. Upon the arrival of these updates, each process produces a list of pages being updated, and then, incorporates it in a list of dirty pages, once additional coherence information piggybacked with a barrier release message has arrived. As in the case of migratory shared data, this list of shared memory page status is flushed to disk at the next synchronization.

3.3.2 Recovery Process

When a failure is detected, the *in_recovery* flag is set and coherence enforcement information (i.e., write-invalidation notice and the list of pages to be updated) is read from the local disk, eliminating the need to produce synchronization messages again. Once the recovery process passes the synchronization point (either a lock or a barrier), it handles memory misses as in the case of failure-free execution. Specifically, a page fault during the recovery process then generates a memory update request to the manager of that shared memory page, and updates its local copy of the page with up-to-date data (i.e., a diff for MW page or a whole page for SW page) received from the manager.

4. Experimental Methodology

For accurate assessment of logging overhead and crash recovery performance, we have integrated message logging (ML) and our adaptive logging (AL) into TreadMarks-based ADSM [2, 15]. We measured the execution time of ADSM without logging protocol incorporated, and utilized it to quantify performance overhead caused by the ML and our AL protocols. As in earlier work, no checkpointing is done in the experiments since the goal is to measure the overhead of logging, and crash recovery performance is obtained by running parallel applications completely to the end and then rolling back node 0, which then begins the recovery process from the log [14, 19].

Our experimental platform is a network of eight SUN Ultra-5 workstations running Solaris 2.6. Each machine contains a 270MHz UltraSPARC-IIi processor, a 256 KB secondary cache, and a 64 MB memory. We also allocated 2G bytes of each local disk for virtual memory paging (i.e., swap space). The size of a virtual memory page is 8K bytes. The network for connecting these machines is a 100Mbps, full-duplex fast Ethernet switch. ADSM implementation is based on TreadMarks, a state-of-the-art software DSM [3]. TreadMarks adopts the UDP/IP protocol for interprocess communication, uses the SIGIO signal for delivering request messages, and relies on the virtual memory trap (SIGSEGV) for invoking the memory coherence enforcement mechanism. All message logs are stored in non-volatile storage (i.e., local disk) for recovery.

Program	Data Size	Synchronization
3D-FFT	100 iterations on $2^7 \times 2^7 \times 2^7$ data	barriers
CG	200 iterations on 14000^2 sparse matrix	barriers
IS	175 ranking of 2^{15} keys	locks and barriers
MG	200 iterations on $2^7 \times 2^7 \times 2^7$ grid	barriers
SOR	100 iterations on $5\mathbf{K} \times 5\mathbf{K}$ input	barriers

Table 1. Applications' Characteristics.

In this study, we employ five parallel applications as benchmark programs, among which 3D-FFT, Conjugate Gradient (CG), Integer Sort (IS), and Multigrid (MG) come

from the NAS benchmark suite [5]. SOR implements a red-black successive over-relaxation and is distributed together with TreadMarks. These applications have been selected in several previous studies of software DSM, e.g., [2, 14, 15, 19]. Table 1 lists application characteristics, including the data size used and the types of synchronization employed.

5. Experimental Results

The performance comparison of two logging protocols, message logging (ML) and our adaptive logging (AL), is made with respect to performance of ADSM without any logging protocol incorporated. We measured disk usage, memory overhead, and total execution time to quantify the overhead of each logging protocol using those five parallel application programs. The crash recovery speeds of the ML-based and our AL-based recovery are also presented.

5.1 Disk Consumption

Table 2 provides data on disk consumption for each of the five applications under ML and our AL protocols. It shows both the number of message logs and the amount of logged data incurred by each protocol when incorporated in ADSM. As can be seen, the AL protocol logs only a small amount of data, ranging from 0.3MB to 5.3MB, whereas the ML protocol produces a much larger total log size, ranging from 23MB to 507MB. This is because our AL protocol leverages the coherence data maintained by ADSM. In particular, the AL protocol does not store data that can be retrieved from the sender process during recovery, while ML simply keeps all incoming data that affect the shared memory status.

Program	Message Logging (ML)		Adaptive Logging (AL)	
	# of Mesg Logged	Log Size (MB) per Process	# of Mesg Logged	Log Size (MB) per Process
3D-FFT	92687	357	374	0.7
CG	367171	507	27656	5.3
IS	12566	39	2760	0.5
MG	103933	292	11211	2.6
SOR	6365	23	476	0.3

Table 2. Disk Consumption of Logged Data.

The ML protocol also leads to a larger number of messages logged, ranging from 6365 to 367171 messages, while our AL protocol keeps considerably fewer messages, ranging from 374 to 27656 messages. All incoming messages that change the state of shared memory in ADSM are logged under the ML protocol, whereas the AL protocol selects to keep only those messages that are necessary for correct recovery. Messages logged under the ML protocol in ADSM include a write-invalidation notice at each synchronization point (i.e., lock and barrier), a summary of modifications (i.e., diff), and an up-to-date copy of each shared memory page. By leveraging the coherence data of ADSM, the AL protocol requires to log only write-invalidation notices since both diff and update data can be retrieved from the sender process during recovery. This substantial reduction in both the number of messages logged and the total size of logged data is due to the fact that the arrival of write-invalidation notices is not as frequent as the arrival of diff and update messages, and the size of a write-invalidation notice is typically smaller than a diff or an update.

5.2 Memory Overhead

The total log size reduction of the AL protocol comes with higher memory overhead under the SW protocol (i.e., under the MW protocol, shared memory pages of ADSM have a summary of modifications stored in the memory of the sender process). For shared memory pages under the SW protocol, our AL protocol creates a volatile log of a shared memory page at the sender process before it sends out such an up-to-date copy of a shared memory page to another process. From our experiment, additional memory overhead due to our AL protocol is a fraction of memory overhead due to coherence enforcement protocols of ADSM. For example, the AL protocol introduces only 1% additional memory overhead under CG and MG (which use the MW protocol), it is about 11% under 3D-FFT and IS (which adopt the SW protocol), and it amounts to roughly 33% under SOR (which also utilizes the SW protocol). It should be noted that, while our AL protocol gives rise to additional memory overhead on top of that overhead due to ADSM coherence enforcement, the additional memory overhead caused by our AL protocol has little impact on ADSM performance. This is because the AL protocol utilizes memory solely for recording update messages (sent out to other processors) which are never read nor modified again, until the recovery process. When more room is needed later to hold such messages, the memory contents are simply swapped out to disks and never swapped back into memory again, unlike the coherence data maintained by ADSM, which are read and modified frequently to realize coherence enforcement, and which are swapped back into memory whenever such references are needed, contributing to execution overhead. The volatile log of our AL protocol thus leads to little penalty in ADSM performance, as will be demonstrated in the next subsection.

While there is no immediate need to do a garbage collection on the volatile log of our AL protocol (since it does not affect ADSM performance), we limit our volatile log size to avoid it from growing unbounded. Garbage collection associated with our AL protocol is not expensive, as we minimize the impact of garbage collection overhead by integrating our garbage collection for the volatile log with the storage reclamation routine of ADSM, which requires excessive message exchange among processors. The cleanup process of our volatile log is therefore overlapped with interprocess communications existing in the storage reclamation routine of ADSM, adding no overhead to ADSM.

5.3 Total Execution Time

Figure 3 depicts the impacts of the ML protocol and our AL protocol on ADSM performance in terms of the total execution time. The total execution time of ADSM without any logging protocol incorporated serves as a performance baseline for comparison. From the data obtained, our AL protocol leads to slight execution time overhead, ranging from 2% to 10% only. This low overhead results directly from a small log size and a low disk access frequency. By contrast, the ML protocol increases the execution time drastically by many folds for every application. Such high overhead is due to recording indiscriminately every incoming message that changes the state of shared memory pages, like a write-invalidation notice, which piggybacked with a lock grant or a barrier synchronization message, a summary of modifications (i.e., a diff), or an up-to-date copy of each shared memory page. By leveraging the coherence data maintained by ADSM, our AL protocol causes only slight execution time overhead and makes the inclusion of fault-tolerant capability in ADSM feasible and attractive.

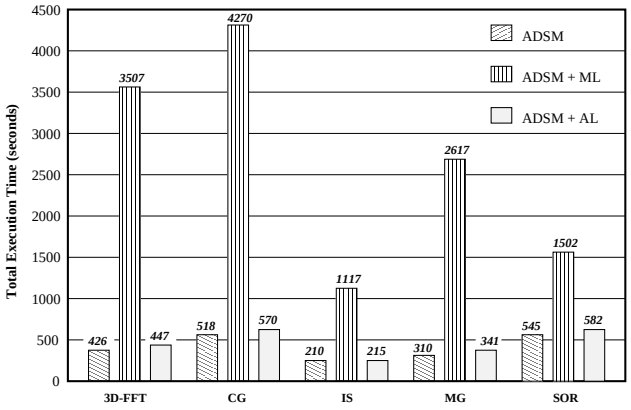

Figure 3. Impacts of Logging Protocols on ADSM Performance.

5.4 Recovery Performance

Figure 4 shows the performance of ML-based recovery and our AL-based recovery after node 0 fails and is rolled back to start recovery from the initial state. We employ the crash recovery time of ADSM without any logging protocol (i.e., Re-Execution) as a performance baseline in this comparison. According to the data presented, ML-based recovery and our AL-based recovery are able to save considerably execution time during the recovery process, in most parallel applications examined. Specifically, ML-based recovery reduces the recovery time by 40% for 3D-FFT, by 28% for CG, by 33% for IS, and by 28% for MG; it slightly increases the recovery time for SOR (i.e., 10% increase), where the computation to communication ratio is very high. This stems from the fact that without the checkpointing protocol incorporated, all computation has to be repeated. In the case of SOR, the overhead of reading logged data from a local disk in ML-based recovery outweights the benefits due to synchronization message elimination and memory miss idle time reduction. For our AL-based recovery, it consistently outperforms ML-based recovery, reducing the recovery time by 52% in 3D-FFT, by 42% in CG, by 49% in IS, and by 45% in MG, when compared with re-execution. It does not shorten the recovery time of SOR. Our AL-based recovery also benefits from synchronization message elimination and memory miss idle time reduction (due to lighter traffic over the network during recovery).

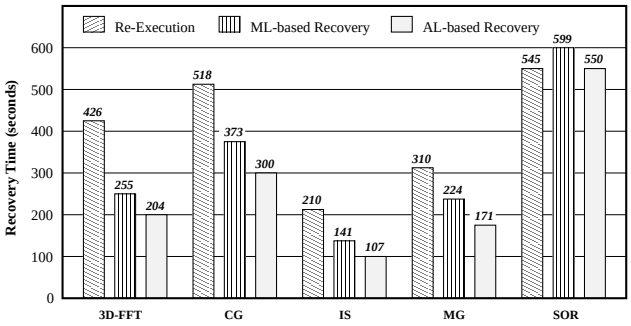

Figure 4. Recovery Performance Shown using Normalized Execution Time.

6. Conclusions

We have dealt with a new, efficient logging protocol for adaptive software DSM, called ADSM, in this paper. The experimental outcomes demonstrate that our adaptive logging (AL) protocol incurs low failure-free overhead, roughly 2% to 10% of ADSM normal execution time, and that our AL-based recovery reduces the recovery time by 42% to 52%, except for SOR (whose computation to communication ratio is very high). This results directly from taking advantage of coherence data managed by ADSM so as to keep only information nonexistence for ADSM coherence but indispensable for fast and correct recovery. Our AL protocol and AL-based recovery are readily applicable to arrive at recoverable ADSM systems effectively.

Acknowledgements

The authors would like to thank Luiz Rodolpho Monnerat, Ricardo Bianchini, and the TreadMarks group at Rice University, for helping us with understanding their adaptive software DSM implementation. This work was supported in part by NSF under Grants CCR-9803505 and EIA-9871315.

References

[1] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[2] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of the IEEE*, pages 467–475, March 1999.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[4] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, February 1997.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA, January 1991.

[6] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault-Tolerance. In *Proc. of the 9th ACM Symp. on Operating Systems Principles (SOSP'83)*, pages 90–99, October 1983.

[7] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation (OSDI)*, pages 59–73, October 1996.

[8] S. Dwarkadas, P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pages 244–255, May 1993.

[9] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proc. of the 24th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-24)*, pages 298–307, June 1994.

[10] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[11] D. B. Johnson and W. Zwaenepoel. Sender-based Message Logging. In *Proc. of the 17th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-17)*, pages 14–19, July 1987.

[12] T. T-Y. Juang and S. Venkatesan. Crash Recovery with Little Overhead. In *Proc. of the 11th Int'l Conf. on Distributed Computing Systems (ICDCS-11)*, pages 454–461, May 1991.

[13] P. J. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[14] A. Kongmunvattana and N.-F. Tzeng. Lazy Logging and Prefetch-Based Crash Recovery in Software Distributed Shared Memory Systems. In *Proc. of the 13th Int'l Parallel Processing Symp. (IPPS'99)*, pages 399–406, April 1999.

[15] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 289–299, February 1998.

[16] T. Park and H. Y. Yeom. An Efficient Logging Scheme for Lazy Release Consistent Distributed Shared Memory Systems. In *Proc. of the 12th Int'l Parallel Processing Symp. (IPPS'98)*, pages 670–674, March 1998.

[17] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proc. of the 12th Int'l Symp. on Reliable Distributed Systems (SRDS-12)*, pages 58–67, October 1993.

[18] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 3(3):204–226, August 1985.

[19] G. Suri, B. Janssens, and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 279–288, June 1995.

[20] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Trans. on Computers*, C-39(4):460–469, April 1990.