Lazy Logging and Prefetch-Based Crash Recovery in Software Distributed Shared Memory Systems

Angkul Kongmunvattana and Nian-Feng Tzeng

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

Abstract

In this paper, we propose a new, efficient logging protocol, called lazy logging, and a fast crash recovery protocol, called the prefetch-based crash recovery (PCR), for software distributed shared memory (SDSM). Our lazy logging protocol minimizes failure-free overhead by logging only data indispensable for correct recovery, while our PCR protocol reduces the recovery time by prefetching data according to the future memory access patterns, thus eliminating memory miss penalty during the recovery process.

We have performed experiments on workstation clusters, comparing our protocols against the earlier reduced-stable logging (RSL) protocol by actually implementing both protocols in TreadMarks, a state-of-the-art SDSM system. The experimental results show that our lazy logging protocol consistently outperforms the RSL protocol. Our protocol increases the execution time slightly by 1% to 4% during failure-free execution, while the RSL protocol results in the execution time overhead of 6% to 21% due to its larger log size and higher disk access frequency. Our PCR protocol also outperforms the widely used simple crash recovery protocol by 18% to 57% under all applications examined.

1. Introduction

Software distributed shared memory (SDSM) provides a shared memory image on top of parallel systems, such as distributed memory machines and workstation clusters. For achieving good performance, SDSM typically adopts relaxed memory consistency models [1] to tolerate potentially high network latencies, realized by delaying communication and coherence enforcement as much as possible. While SDSM continues to improve its performance and scalability, the probability of system failures increases as its size grows. Hence, for SDSM systems to be employed in long-running applications or in high-availability situations, a mechanism for supporting fast crash recovery from node failures is required, giving rise to *recoverable SDSM systems*.

Over the past few years, several log-based rollback-recovery protocols, traditionally called message logging, have been proposed for providing fault tolerance in SDSM systems under relaxed memory consistency model [7, 13, 17]. Specifically, Suri *et al.* proposed the first logging protocol [17] for relaxed memory consistency-based SDSM systems, with each process logging all nondeterministic events (i.e., the arrival of coherence-related messages) in volatile memory and flushing them to stable storage when it has to communicate with another process. This logging scheme allows SDSM to recreate the execution by restarting from the most recent checkpoint and replaying the log of events. Since the execution between any two consecutive nondeterministic events is itself deterministic, it produces an exact replay of the execution before the failure. This type of pessimistic logging, however, incurs a high *failure-free* overhead during normal execution.

To make recoverable SDSM more affordable, it is important to reduce the logging overhead, e.g., memory space and the number of disk accesses due to the flushing of volatile logs. Disk access frequency is critical to SDSM performance because the high disk latency has a potential to lengthen the critical path of program execution. Recently, another logging scheme called reduced-stable logging (RSL) has been considered [13]. It reduces the log size by logging only selected nondeterministic events, such as the arrival of coherence enforcement messages during the synchronization process, but not the arrival of memory update messages, since the latter can be deterministically recreated once coherence enforcement information (i.e., a list of dirty pages) is applied. Their simulation results indicated that RSL utilized less disk space for the logged data than that of [17].

Lazy logging is a new, efficient protocol we propose in this paper for achieving recoverable SDSM, aimed at reducing both the log size and the number of disk accesses. Unlike other protocols, a lazy protocol such as lazy logging does not log every nondeterministic events (like the arrival of coherence enforcement or memory update messages). Instead, it relies on the log of deterministic events, i.e., read

or write accesses of shared data, that cause nondeterministic events essential to correct program execution, i.e., requesting for an up-to-date shared data. Therefore, lazy logging precisely logs the events that cause memory updates, while RSL logs every event that invalidates the memory, no matter whether it induces any memory update or not. In Figure 1, for instance, both processes p_2 and p_3 under the RSL protocol log the list of dirty pages piggybacked on the lock-grant messages received, and p_2 also flushes its volatile logs to stable storage when it sends a message to p_3. With lazy logging, however, p_2 logs nothing because there is no nondeterministic events created in that interval, and p_3 logs only the list of pages needed to be updated (i.e., page x and a for that interval). Obviously, in the worst case, lazy logging will log and flush as much as RSL, but in general, it will create a smaller log size and induce fewer accesses to stable storage. To lower the number of disk accesses further, we also introduce an optimization technique based on the optimistic logging protocol [16]. This technique allows for our logging protocol to control disk accesses resulting from flushing volatile logs. More details on lazy logging protocol and optimization technique are presented in Section 3.

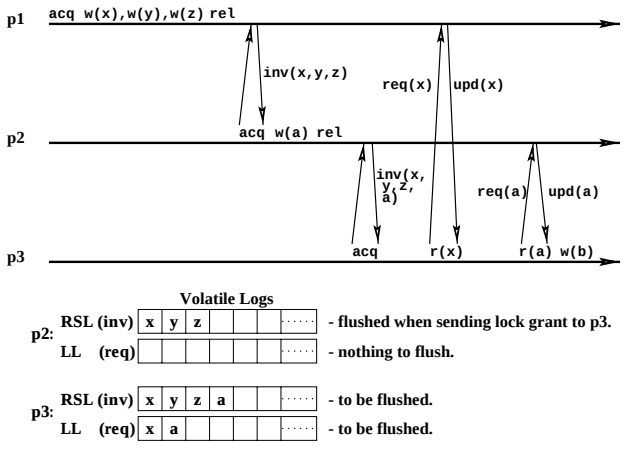

Figure 1. Logging and Flushing Scenarios under Different Protocols.

For fast crash recovery, we also propose in Section 3, a *prefetch-based crash recovery* (PCR) protocol that reduces both the memory miss penalty and the number of messages exchanged during the recovery process. To assess the impacts of our logging and crash recovery protocols on SDSM performance, we implemented both RSL and our protocols in TreadMarks (a state-of-the-art SDSM), which was run on eight Sun Ultra-5 workstations. Experimental results under three parallel applications demonstrate that our lazy logging protocol causes very low failure-free overhead, ranging from 1% to 4% of the normal execution time, and additionally, our prefetch-based crash recovery protocol shortens the recovery time significantly (when compared with other recovery protocols).

The outline of this paper is as follows. Section 2 provides basic background pertinent to this work. Our lazy logging and prefetch-based crash recovery protocols are described in Section 3. Section 4 presents the performance results of our proposed protocols, after stating our experimental setup. We briefly discuss related work in Section 5 and conclude this paper in Section 6.

2. Pertinent Background

Our logging and crash recovery protocols are developed for SDSM under lazy release consistency (LRC). It is therefore essential to understand SDSM and its optimization protocols designed to reduce the performance penalty during failure-free execution. This section provides pertinent background on SDSM and its coherence protocols, including an overview on the log-based rollback-recovery protocol.

2.1 SDSM

Parallel programming on distributed memory machines with explicit message-passing has been known to be difficult. SDSM simplifies this programming task by providing an illusion of shared memory image to the programmers. The implementation of SDSM often relies on virtual memory page protection hardware to initiate the coherence enforcement of a shared memory image. Such hardware support is readily available in most, if not all, commodity microprocessors. While versatile and attractive, SDSM has to minimize its coherence traffic overhead, because its performance is especially sensitive to the traffic amount over the network. To this end, the implementation of SDSM usually adopts a relaxed memory consistency model [1], such as release consistency [8], because of its potentially low coherence traffic.

Release consistency guarantees that shared memory is consistent only after synchronization operations, i.e., locks and barriers. Lazy release consistency (LRC) [11] is an efficient software implementation of release consistency model. It postpones coherence enforcement until the acquire is performed. 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 SDSM generally involves fewer messages and less data exchanged than other SDSM implementations. In this paper, we are interested in page-based SDSM developed under the notion of LRC.

2.2 Log-Based Rollback-Recovery

Log-based rollback-recovery has its root in message-passing systems [4, 10, 16]. It follows the *piecewise deterministic system model* [16]: A process execution is divided into a sequence of deterministic state intervals, each of which starts at the occurrence of a nondeterministic event like a message receipt. The execution between nondeterministic events is completely deterministic. During a failure-free period, each process periodically saves its execution state in stable storage as a checkpoint, and all messages received are logged in the volatile memory before being flushed to stable storage whenever the local process has to send a message to another process. Should a failure occur, the recovery process starts from the last checkpoint and the logged messages are replayed to reconstruct the execution before the failure.

3. Proposed Logging and Recovery

3.1 Lazy Logging

We first introduce a new logging protocol for LRC-based SDSM, called lazy logging, which is aimed at reducing both the logged data amount and the number of disk accesses. This protocol is based on logging deterministic events that cause nondeterministic events truly essential to the consistency of shared memory accesses in SDSM. There are three types of nondeterministic events in SDSM: The arrival of lock grant messages, the arrival of memory update messages, and the arrival of memory update request messages. However, only the first two types of messages received contain memory coherence-related information that will change the status of some local copies of shared memory pages. Specifically, the lock grant message always piggybacks a list of dirty pages to be invalidated, and the memory update message contains up-to-date data for local copies of shared data that are out-of-date, whereas the memory update request message does not have any affect on the coherence status of the local copies of shared data.

While it is easy to simply log both lists of dirty pages and up-to-date data as proposed in [17], its high logging overhead makes this approach unattractive. By determining the relation between these two coherence-related nondeterministic events, one can notice that the arrival of memory update messages is actually dependent on the contents of the previous lock grant message received, i.e., the list of dirty pages. Therefore, it is adequate to log only the list of dirty pages in order to recreate the execution once a failure happens, known as reduced-stable logging (RSL) [13].

After carefully analyzing the memory coherence enforcement mechanism, we found that correct program execution relies solely on the integrity of shared memory accesses made by each process. It is hence unproductive to log the whole list of dirty pages, because the list of such dirty pages that request the up-to-date data alone is sufficient to reconstruct correct execution replay. As a result, our proposed protocol logs only the list of those dirty pages that are accessed by the process: Each process logs the messages sent out to request the up-to-date data in volatile memory and flushes them to stable storage before it communicates with another process. Obviously, the logged data under our proposed protocol is always a subset of RSL logged data; therefore, our protocol at worst logs the same amount of data as the RSL protocol. Experimental results indeed demonstrate that our lazy logging usually logs less data than RSL, by 35% to 96% for parallel applications examined, as will be detailed in Section 4.2.

3.1.1 Optimization

To reduce the number of disk accesses further, we consider to incorporate into our lazy logging protocol an optimization technique based on the optimistic logging protocol [16]. This technique postpones the flushing of volatile logs to stable storage until a specific number of intervals has elapsed. The specific number of intervals is defined as the *flushing distance*, denoted by f_d. Since the process advances the execution interval (by incrementing its local vector timestamp) only at the synchronization points, f_d = 1 represents the case of pessimistic logging, flushing every time it sends out a synchronization message. By adopting this optimization technique, we allow for the proposed logging protocol to control disk access frequency, which is critical to the failure-free overhead. Should a failure occur, the volatile logs will be lost and cannot be used during the recovery process. Consequently, the recovery process has to restart from the last checkpoint and reconstruct the execution by recreating all nondeterministic events, instead of obtaining coherence-related information from the logged data. However, unlike the rollback-recovery process of optimistic logging in message-passing systems, the surviving processes in SDSM need not to be rollbacked because there is no process that becomes an orphan after a failure.

Upon a failure in SDSM, there is no orphan process created even in the presence of optimistic logging-based optimization due to the following reasons. While three types of outgoing messages exist in SDSM, none of them can create an orphan process since both lock grant and memory update messages will be sent out only in response to the lock and memory update request messages from other processes; they will not be recreated during the recovery process. Memory update request messages do not cause any change in the memory coherence status of the receiving process, and therefore it is unnecessary for surviving processes to rollback and “unreceive” this type of messages. As a result, no orphan process is created.

3.1.2 Checkpointing

To limit the amount of work that has to be repeated after a failure, each process also periodically takes a checkpoint. Our proposed logging protocol allows for each process to independently perform asynchronous checkpointing. A checkpoint consists of all local and shared memory contents, the state of execution, and the internal data structure used by SDSM. All such information is needed at the beginning of a crash recovery process to avoid restarting from the (global) initial state. Since our logging protocol never causes any surviving process to rollback after a failure, only the most recent checkpointing is needed for the recovery process. Therefore, after a checkpoint is completed, each process may discard its previous checkpoint together with all logged data.

3.2 Prefetch-Based Crash Recovery

Traditionally, when a process failure is detected, a recovery process is invoked to reconstruct the execution prior to the failure by restarting from the most recent checkpoint and replaying the log of events. For SDSM, the log of events contains lists of dirty pages to be invalidated or updated at each synchronization point to preserve memory consistency. The widely adopted simple crash recovery (SCR) protocol always performs invalidation because it does not want to unnecessarily update all dirty pages (as it may or may not be needed). While this recovery protocol is straightforward for implementation, it often gives rise to a mediocre crash recovery speed.

To improve crash recovery performance, we propose a new protocol, referred to as a prefetch-based crash recovery (PCR) protocol, for fast recovery. It uses the knowledge of future shared memory access patterns, gathered during the failure-free execution by our lazy logging protocol, to bring a remote data into local memory before it is actually needed. This data prefetch is performed at each synchronization point by sending a single memory update request message to each participating process that holds an up-to-date data. Upon receiving the memory update message, PCR uses the logged data to identify whether the accesses to the shared memory page involve any write operation: if not, the page status is set to read-only; otherwise, a twin, pristine copy of that page is created and the page status is set to read-write, satisfying multiple-writer protocol [6] requirements. As a result, PCR eliminates both page faults and idle time resulting from shared memory access misses during the recovery process. Figure 2 shows the differences of the coherence enforcement actions involved during failure-free execution and the recovery processes under SCR and PCR protocols. In this example, process p_3 is crashed some time after the volatile logs of this interval are flushed to stable storage, but before the next checkpoint is created. After a failure is detected, the recovery process starts from the last checkpoint

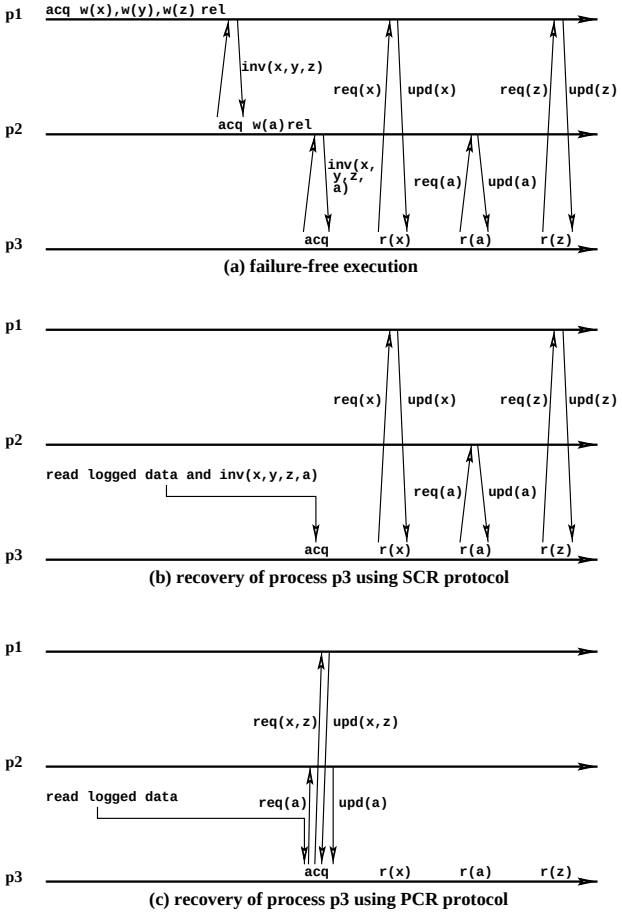

Figure 2. Coherence Enforcement Involved during Normal Execution and Recovery.

and replays the logged data. During this snapshot interval, the recovery process of p_3 has to request for up-to-date data of three shared memory pages: x, a, and z. Under the SCR protocol depicted in Figure 2(b), the update of these three pages causes three page faults and generates six messages, whereas PCR shown in Figure 2(c) involves only four messages and exhibits no page fault. Experimental results confirm that this reduction of messages exchanged and elimination of memory misses significantly improve crash recovery performance, as will be demonstrated in Section 4.3.

3.3 Implementation

Figure 3 shows the pseudo code of the SDSM routines involved in the logging and crash recovery processes. During the failure-free execution, volatile logs are created when a process requests the up-to-date data and are flushed to stable storage at each point of barrier synchronization or lock acquire. When a failure is detected the *in_recovery* flag is set and up-to-date data from remote processes are prefetched at the synchronization point. Therefore, no page fault arises during the recovery process.

Barrier

```
if (in_recovery)
   read list of dirty pages from log;
   send diff request to any process
        with up-to-date copy;
   apply diffs received to local pages;
   if (page_will_be_written)
      create twin of page;
      set page status to read-write;
   else
      set page status to read-only;
   endif
else
   if (barrier_manager)
      flush log to stable storage;
      wait for all other processes
           to check-in;
      invalidate dirty pages;
      send barrier release message
           piggybacked with list of
           dirty pages to all other
           processes;
   else
      send check-in message to
           barrier manager;
      flush log to stable storage;
      wait for barrier release message;
      invalidate dirty pages;
   endif
endif
```

Lock Acquire

```
if (in_recovery)
   read list of dirty pages from log;
   send diff request to any process
        with up-to-date copy;
   apply diffs received to local pages;
   if (page_will_be_written)
      create twin of page;
      set page status to read-write;
   else
      set page status to read-only;
   endif
else
   send lock request to lock manager;
   flush log to stable storage;
   wait for lock grant message;
   invalidate dirty pages;
endif
```

Page Fault Handler

```
send diff request to any process
     with up-to-date copy;
log diff request content to volatile log;
apply diffs received to local pages;
if (write_fault)
   create twin of page;
   set page status to read-write;
else
   set page status to read-only;
endif
```

Figure 3. Pseudo Code for Lazy Logging and Prefetch-based Crash Recovery.

4. Performance Evaluation

In this section, we briefly describe hardware and software employed in our experiments and also present the evaluation results of our proposed protocols for recoverable SDSM. We first compare the performance results of our proposed logging protocol with those of the RSL protocol and then identify overhead involved using the normalized execution time breakdown. Since the goal is to compare the logging overhead, no checkpoint is created in our experiments. Subsequently, we also evaluate and contrast the crash recovery speeds of our PCR and the SCR protocols.

4.1 Experiment Setup

Our experimental platform is a collection of eight Sun Ultra-5 workstations running Solaris version 2.6. Each workstation contains a 270 MHz UltraSPARC-IIi processor, 256 KB of external cache, and 64 MB of physical memory. These machines are connected via a fast Ethernet (of 100 Mbps). We allocated 512 MB of the local disk at each workstation for virtual memory paging and left 1.2 GB of local disk space available for logged data.

We evaluated our proposed logging and crash recovery protocols by incorporating them in TreadMarks [2], a LRC-based SDSM. For the measurements of failure-free overhead and the crash recovery speed, the experiments were performed on three parallel applications included in TreadMarks distribution: 3D-FFT, SOR, and Water. 3D-FFT is a 3-dimensional fast fourier transform originally from the NAS benchmark suite [3]. SOR implements the red-black successive over-relaxation algorithm for solving discretized Laplace equations. Water is a molecular dynamics simulation from the SPLASH benchmark suite [15]. The data set size and synchronization type of each application used in this experimental study are listed in Table 1.

Program	Data Set Size	Synchronization
3D-FFT	400 iterations on $2^9 \times 2^9 \times 2^9$ data	barriers
SOR	350 iterations on 2K$\times$2K input	barriers
Water	100 iterations on 512 molecules	locks and barriers

Table 1. Applications' Characteristics.

4.2 Performance Results of Logging Protocols

Table 2 presents the failure-free overhead of logging protocols. It lists total execution time, percentage overhead in comparison to the execution time of the program with no logging protocol incorporated, the log size, and the number of times the volatile logs are flushed to stable storage. Since no logging exists in the original TreadMarks, its execution time is used as a performance base line. Both our lazy logging and the RSL protocols provide TreadMarks with crash recovery capabilities, but they involve different performance penalty amounts. From Table 2, it is apparent that our lazy logging consistently results in lower failure-free overhead than RSL, which requires a larger log size and more frequent accesses to stable storage. The log size of our protocol is smaller than the RSL protocol by as much as 35% for 3D-FFT, 96% for SOR, and 68% for Water. Lazy logging also reduces the number of disk accesses through delaying the flushing operation. By setting the flushing distance of our lazy logging to ten (i.e., f_d = 10) in this experiment, we lower the disk access frequency by as much as 89% for 3D-FFT and SOR, and by more than 90% for Water.

Logging Protocol	Execution Time (sec.)	Percentage Overhead	Log Size (KB)	# of Flushes
None	83.5	-	-	-
Lazy Logging	84.4	1.1	383	80
RSL	101.3	21.3	592	804

(a) 3D-FFT

Logging Protocol	Execution Time (sec.)	Percentage Overhead	Log Size (KB)	# of Flushes
None	445.2	-	-	-
Lazy Logging	452.1	1.5	19	70
RSL	471.6	5.9	512	703

(b) SOR

Logging Protocol	Execution Time (sec.)	Percentage Overhead	Log Size (KB)	# of Flushes
None	125.0	-	-	-
Lazy Logging	129.7	3.8	299	55
RSL	150.2	20.2	944	702

(c) Water

Table 2. Overhead Details under Different Logging Protocols.

Figure 4 shows the impacts of different logging protocols on execution time. For each application, the top-most portion of the normalized execution time for our lazy logging and the RSL cases (labeled as "Logging Overhead") represents failure-free overhead associated with performing logging. The remaining time is broken down into the following four categories, from top to bottom: time spent in stalled waiting for (i) synchronization and (ii) remote memory misses, respectively; (iii) time spent in executing SDSM protocols; and (iv) time spent in computation and the OS kernel. According to Figure 4, our lazy logging protocol adds very little overhead to the execution time, ranging from 1% to 4% only. This low overhead results directly

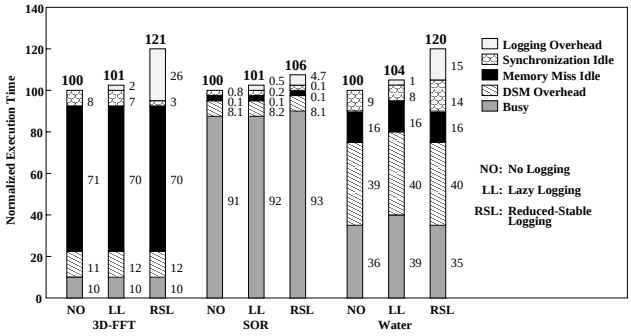

Figure 4. Impacts of Logging Protocols on Execution Time.

from a small log size and a low disk access frequency. By contrast, RSL increases the execution time by as much as 21% for 3D-FFT, 6% for SOR, and 20% for Water. RSL leads to reduced synchronization idle times for 3D-FFT and SOR, because flushing volatile logs is done during the busy-waiting period of the synchronization operation, but not for Water since its busy-waiting period is not long enough to accommodate the flushing operation; as a result, it even prolongs the synchronization idle time in that application. Note that busy-waiting period is highly dependent on the application code, overall system load and its distribution, and inputs. We also notice that the logging protocols have slightest impacts on SOR performance. This is because the computation-to-communication ratio in SOR is very high, and therefore logging overhead caused by message logging becomes insignificant.

4.3 Performance Results of Crash Recovery

Table 3 lists the measurements of crash recovery performance in terms of recovery time, the number of synchronization messages, the number of data request messages, and the number of page faults. Both synchronization and data request messages represent traffic over the network, while page faults contribute to the memory miss idle time, SDSM overhead, and time spent in the OS kernel. As no crash recovery protocol is incorporated in the original TreadMarks, should a failure occur, it has to restart from the (global) initial state without any logged data for execution replay. As a result, it spends the same amount of time for re-executing the program without any time savings. This normal execution time is employed as a base line for crash recovery performance comparison. Under our prefetch-based crash recovery (PCR) protocol, the recovery time is shortened by as much as 83% for 3D-FFT, 8% for SOR, and more than 65% for Water (see Table 3). Although the page fault has been eliminated and the number of messages exchanged has been minimized for the SOR application, we are unable to significantly shorten its crash recovery time, because computation itself dominates the overall execution time, as shown in Figure 5. While the simple crash recovery

(SCR) protocol lowers the recovery time by eliminating the synchronization messages, it increases recovery time due to the memory misses and reading large logged data, where high overhead results. From Table 3, SCR yields a net reduced recovery time by 59% for 3D-FFT and 49% for Water, but it actually increases the recovery time by more than 45% for SOR, where overhead associated with the protocol outweights its gain. Hence, our PCR protocol outperforms the SCR protocol by 18% to 57% under parallel applications examined.

Recovery Protocol	Recovery Time (sec.)	# of Sync. Messages	# of Data Req. Mesg.	# of Page Fault Traps
None	83.5	33691	378	354
PCR	14.4	0	147	0
SCR	34.4	0	378	354

(a) 3D-FFT

Recovery Protocol	Recovery Time (sec.)	# of Sync. Messages	# of Data Req. Mesg.	# of Page Fault Traps
None	445.2	9062	140	3219
PCR	410.5	0	140	0
SCR	664.7	0	140	3219

(b) SOR

Recovery Protocol	Recovery Time (sec.)	# of Sync. Messages	# of Data Req. Mesg.	# of Page Fault Traps
None	125	109239	2946	600
PCR	42	0	2769	0
SCR	64	0	2946	600

(c) Water

Table 3. Overhead Details under Different Crash Recovery Protocols.

Figure 5 illustrates the impacts of crash recovery protocols on recovery performance. For each application, the top-most portion of the normalized recovery times for our PCR and the SCR cases (labeled as "Recovery Overhead") represents recovery overhead. The remaining time components are the same as in Figure 4. According to the figure, our PCR eliminates both synchronization and memory misses idle times completely through using the logged data and performing data prefetches, respectively. Data prefetching in PCR not only avoids the page faults, but also decreases the time spent in the OS kernel and SDSM protocols. While SCR eliminates the synchronization idle time by reading the logged data, the recovery time may actually increase if the log size is large, as the recovery process then has to spend a long time in reading it. Reduction in memory miss idle time under SCR is due to lighter traffic over the network during the recovery process. In summary, PCR always outperforms SCR because it involves very low recovery overhead and alleviates most of the SDSM performance bottleneck due to high network latencies.

5. Related Work

Most previous work on rollback-recovery in SDSM adopted synchronous checkpointing protocols to establish a consistent global state after a failure [5, 9, 12, 18]. While

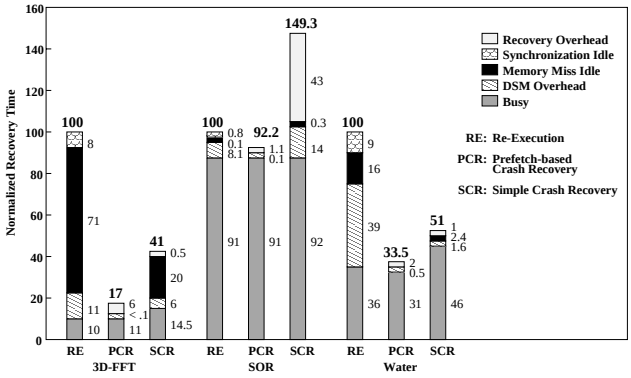

Figure 5. Impacts of Crash Recovery Protocols on Recovery Performance.

those protocols are not susceptible to an unbounded rollback, attributed to the *domino effect*, their failure-free overheads are often high because all processes have to synchronize during the checkpoint process.

To this end, log-based rollback-recovery is a preferred choice for implementing recoverable SDSM, thanks to its low failure-free overhead. Richard and Singhal [14] considered logging and asynchronous checkpointing protocols for sequentially consistent SDSM. Their logging protocol logs the contents of all shared memory accesses and causes high overhead due to a large log size and high disk access frequency. Suri, Janssens, and Fuchs [17] reduced the overhead of that protocol by logging only the records of all shared memory accesses, instead of their contents. They also proposed a logging protocol for SDSM under relaxed memory consistency models [17], realized by logging all received coherence-related messages in volatile memory and flushing them to stable storage before communicating with another process. This protocol, however, suffers from communication-induced accesses to stable storage. Costa *et al.* introduced a vector clock logging protocol [7], based upon the idea of sender-based message logging protocols [10], in which the logged data are kept in volatile memory of the sender process. Unfortunately, the protocol cannot handle multiple-node failures. Recently, Park and Yeom [13] described a logging protocol known as reduced-stable logging, obtained through refining the protocol proposed in [17] by logging only the content of lock grant messages, i.e., the list of dirty pages. Their simulation results indicated that the protocol utilized less disk space for the logged data.

In this paper, we have introduced a new logging protocol dubbed lazy logging, which significantly reduces the log size and the number of accesses to stable storage. We have collected the performance data of our protocol and the reduced-stable logging protocol on a workstation cluster using TreadMarks. Additionally, we also introduce a prefetch-based crash recovery protocol to speed up the recovery process by reducing the number of messages exchanged and

eliminating the memory misses; this is the first recovery protocol actually implemented on a real system for evaluation. No prior work has ever addressed effective recovery, despite it is important to a recoverable SDSM system.

6. Conclusions

We have dealt with a new, efficient log-based rollback-recovery technique for recoverable SDSM implementation in this paper. The experimental outcomes demonstrate that our lazy logging protocol incurs very low failure-free overhead, roughly 1% to 4% of normal execution time, and that our prefetch-based crash recovery protocol improves crash recovery speed by 18% to 57% when compared with the widely used simple crash recovery protocol. This results directly from our lazy logging, which keeps only information essential for recovery to a consistent execution state and requires fewer disk accesses. While a simple crash recovery protocol follows the normal execution replay and invalidates shared memory at all synchronization points, our proposed recovery protocol totally eliminates the memory miss idle and obviates the need of memory invalidation steps through prefetching an up-to-date data before it will be accessed in a subsequent execution interval, therefore giving rise to faster crash recovery. Our lazy logging and prefetch-based crash recovery are readily applicable to arrive at recoverable SDSM systems effectively.

Acknowledgements

This material was based upon work supported in part by the 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, 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.

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

[4] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault-Tolerance. In *Proc. of the 9th ACM Symp. on Operating Systems Principles*, pages 90–99, Oct. 1983.

[5] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *Proc. of the 14th Int'l Symp. on Reliable Distributed Systems*, pages 96–105, Sep. 1995.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[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*, Oct. 1996.

[8] 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.

[9] G. Janakiraman and Y. Tamir. Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers. In *Proc. of the 13th Int'l Symp. on Reliable Distributed Systems*, pages 42–51, Oct. 1994.

[10] 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.

[11] 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.

[12] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 289–298, June 1995.

[13] 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)*, March 1998.

[14] 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*, pages 58–67, 1993.

[15] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

[17] 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.

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